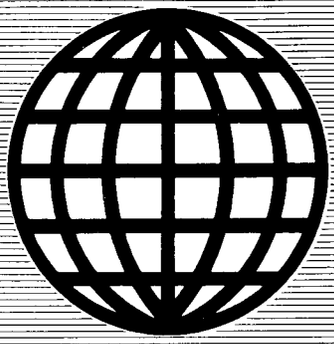


The COMPUTER JOURNAL®

Programming - User Support
Applications

Issue Number 42 January / February 1990 $3.00

Dynamic Memory Allocation

Using BYE with NZCOM

C and the MS-DOS Screen

Lists and Object Oriented Forth

The Z-System Corner

68705 Embedded Controller Application

Advanced C/PM

The NS 32000

The Computer Corner

ISSN # 0748-9331

The Computer Journal

Editor/Publisher
Art Carlson

Art Director
Donna Carlson

Circulation
Donna Carlson

Contributing Editors
Bill Kibler
Bridger Mitchell
Clem Pepper
Richard Rodman
Jay Sage
Dave Wenstein

The Computer Journal is published six times a year by Technology Resources, 190 Sullivan Crossroad, Coulmbia Falls, MT 59912
(406) 257-9119

Entire contents copyright © 1990 by Technology Resources.

Subscription rates—$16 one year (6 issues), or $28 two years (12 issues) in the U.S., $22 one year in Canada and Mexico, and $24 (surface) for one year in other countries. All funds must be in U.S. dollars on a U.S. bank.

Send subscription, renewals, address changes, or advertising inquires to: The Computer Journal, 190 Sullivan Crossroad, Columbia Falls, MT 59912, phone (406) 257-9119.

The COMPUTER JOURNAL

Issue Number 42 January / February 1990

Editorial 3

Dynamic Memory Allocation 4
Techniques for allocating memory at run time with examples in Forth. By Dreas Nielsen.

Using BYE with NZCOM 11
Getting BYE and NZCOM to peacefully coexist is not easy—here's how to do it. By Chris McEwen.

C and the MS-DOS Screen Character Attributes .. 15
How to talk to the screen with C. By Clem Pepper.

Forth Column 21
Lists and object oriented Forth.
By Dave Weinstein.

The Z-System Corner 24
Genie Roundtable discussions, BDS Z, and a review of some Z-System fundamentals.
By Jay Sage.

68705 Embedded Controller Application 29
An example of a single-chip microcontroller application. By Joe Bartel.

Advanced CP/M 31
PluPerfect Writer and using BDS C with REL files.
By Bridger Mitchell.

Real Computing 34
The NS 32000. By Richard Rodman.

Computer Corner 40
By Bill Kibler.

Plu*Perfect Systems == World-Class Software

BackGrounder ii **$75**

Task-switching ZCPR34. Run 2 programs, cut/paste screen data. Use calculator, notepad, screendump, directory in background. CP/M 2.2 only. Upgrade licensed version for $20.

Z-System **$69.95**

Auto-install Z-System (ZCPR v 3.4). Dynamically change memory use. Order **Z3PLUS** for CP/M Plus, or **NZ-COM** for CP/M 2.2.

PluPerfect Writer **$35**

Powerful text and program editor with EMACS-style features. Edit files up to 200K. Use up to 8 files at one time, with split-screen view. Short, text-oriented commands for fast touch-typing: move and delete by character, word, sentence, paragraph, plus rapid insert/delete/copy and search. Built-in file directory, disk change, space on disk. New release of our original upgrade to Perfect Writer 1.20, now for all Z80 computers. On-disk documentation only.

ZSDOS ..... **$75, for ZRDOS users just $60**

Built-in file DateStamping. Fast hard-disk warmboots. Menu-guided installation. Enhanced time and date utilities. CP/M 2.2 only.

DosDisk ..... **$30 - $45**

Use MS-DOS disks without copying files. Subdirectories too. Kaypro w/TurboRom, Kaypro w/KayPLUS, MD3, MD11, Xerox 820-I w/Plus 2, ON!, C128 w/1571 -- $30. SB180 w/XBIOS -- $35. Kit -- $45. Kit requires assembly language expertise and BIOS source code.

MULTICPY **$45**

Fast format and copy 90+ 5.25" disk formats. Use disks in foreign formats. Includes DosDisk. Requires Kaypro w/TurboRom.

JetFind **$50**

Fastest possible text search, even in LBR, squeezed, crunched files. Also output to file or printer. Regular expressions.

To order: Specify product, operating system, computer, 5 1/4" disk format. Enclose **check**, adding $3 shipping ($5 foreign) + 6.5% tax in CA. Enclose invoice if upgrading BGii or ZRDOS.

Plu*Perfect Systems
410 23rd St.
Santa Monica, CA 90402
(213)-393-6105 (eves.)

BackGrounder ii ©, **DosDisk** ©, **Z3PLUS** ©, **PluPerfect Writer** ©, **JetFind** ©
Copyright 1986-88 by Bridger Mitchell.

Editor's Page

Production Changes

The production of a magazine involves a lot of time—both calendar time and people hours. We have been planning for production (printing and binding) changes which will enable us to get TCJ out on schedule, with a shorter lead time, while freeing up more of our time for development.

Part of the change involved working with a new vendor, and we sent the camera ready copy out on schedule. Then we waited. And we waited some more. Finally, after the issue was scheduled to have been shipped, we canceled and got the artwork back.

Now, after losing about four weeks, we are going back to the previous vendor. They hadn't left room in their schedule for TCJ, so we'll have to squeeze it in. What a mess!

This issue will arrive very late, but we are planning on following with #43 back on the normal schedule.

Microcontrollers

There are many situations where a dedicated microcontroller is a better choice than a microcomputer--but there is very little application level information available. The design of microcontroller applications is one of the long-delayed projects I will be working on.

I divide controller applications into three categories, but they are not all inclusive and there is a lot of overlap. One category is Independent, where the device has no connections to another device. An example of this could be a hand-held unit which measures and records air temperature and velocity (wind speed), then displays temperature, velocity, and the wind chill factor. This could be used to determine the potential dangers of exposure for outdoors workers and sportsmen. It could even be carried by a downhill skier so that a resort could evaluate the potential dangers for their guests. A further enhancement could include thermosensors on the skier's skin (both front and back) to evaluate ski apparel--perhaps the material should be thicker and more wind resistant on the front because that is the prevailing direction of the air flow.

A second category is Wired-Remote, where the device communicates with a host system. Although I use the term "Wired," the communications can be carried out by wire, radio, optical, or other means. An example of this could be a unit which monitors refrigeration systems, recording time, temperature, door openings, etc. It would retain this data for uploading to the host system upon interrogation, but would generate an alert signal for an out of control situation. The host system would not have to poll the cooling systems to determine if they were within limits because the remote would alert it to any problems.

A third category would be buss or daughter board units where the device communicates directly through the buss. Devices in this category would be platform-dependent. The busses considered for now will be PC/AT, STD, S-100, and custom busses. Depending on the configuration, the host system could be a normal microcomputer or another microcontroller.

I am interested in distributed and parallel processing using multiple controllers for real-world measurement and control applications. I am not comfortable with using one CPU (regardless of how fast or powerful it is) for multitasking real time control. Much of the initial work will involve developing guidelines for selecting between a microcomputer and a microcontroller, and determining the most suitable processor. Application requirements, size, environmental conditions, hardware and software development costs (and time), production costs, production quantity, and market conditions will all be considered.

Evaluating and designing the application will often involve at least as much time and effort as the actual hardware and software design. Since this publication is about computers and not control engineering, I am not sure how much of the application design information should be included. But, knowledge of the application is vital to successful design, and the best employment opportunities are for people who understand both applications and hardware/software. I need your feedback on how much (if any) of this you want to hear about. I will also appreciate information from those now working in this field.

Schematic Drawing

The controller hardware articles will involve a lot of schematics, and my hand drawn schematics are terrible. I just wouldn't prepare hardware articles if I had to hand ink the schematics--I needed a schematic drawing program which would would work with the LaserJet.

I first tried a graphics drawing program, but that was completely unsuitable. Next I tried a Computer Aided Drafting program (Generic CADD) which was good for machine shop type drawings, but was poor for schematics. I am now working with SCHEMA II+ (Omation, Inc., 801 Presidential Drive, Richardson, TX 75081, (214)231-5167).

My requirements are for a program which will enable me to produce working and camera ready schematic drawings. I want something which is powerful enough to save me time, but flexible enough to stay out of my way and let me do things my way. I do not at this time need printed

(Continued on page 30)

Dynamic Memory Allocation

by Dreas Nielsen

Many programs require the ability to manipulate data elements of indeterminate size or number. Text strings are an example of one such type of data: each string may be a different length, and it is usually neither feasible nor economical to statically allocate (at compile or assembly time) a buffer capable of holding the largest possible string. Programs that manipulate arrays of numbers often need to establish their memory requirements dynamically—that is, at run time, without the use of a statically allocated buffer. Creation of linked lists, trees, and other more complex data structures also typically cannot be carried out with statically allocated memory. The solution to this problem is to provide the program with a means to dynamically allocate memory at run time. Dynamically allocated memory is drawn from the pool of main memory remaining after a program has been loaded and the stack (and other language-specific data structures) have been established. Dynamic memory allocation requires more complex run-time support than static buffers (which require none), but provides greater flexibility. If the available memory is to be re-used repeatedly for different purposes, the special-purpose code needed to manipulate a statically allocated block may be equivalent in size and complexity to that for general-purpose dynamic memory allocation.

Dynamic memory allocation is used by many common languages. In some of these, it is solely under the control of the language itself (e.g., strings in BASIC and dBase, all objects in Smalltalk and Lisp), in others partial or complete control is given to the programmer (e.g., Pascal, C, and Modula-2). Explicit control of dynamic memory allocation is a powerful tool, and fundamental to effective implementation of many useful algorithms.

This article provides an introduction to the implementation of dynamic memory allocation, covering a few of the principles and providing examples for illustration. This is a topic that does not seem to be well covered in the literature; indeed, Knuth (1973) and Aho, Hopcroft, and Ullman (1983) seem to be the only commonly available references that treat it in depth. On the other hand, why would you want to know anything more about the implementation of such an arcane feature, particularly if you are not writing a compiler or operating system? First of all, you may need (or want) to use a language that does not intrinsically provide this feature but does have the systems-programming capability to implement it. Secondly, the memory allocation functions provided by a language or standard library may not exactly suit your needs. Curiosity, of course, may be sufficient reason in and of itself.

Despite the relative paucity of information, dynamic memory allocation is not as complex as it may seem. Furthermore, an understanding of the techniques and costs involved can help you decide when a general-purpose routine is suitable, when a specialized routine may be better, and when to do without dynamic memory allocation. Source code for both a simple general-purpose routine and a more efficient specialized routine is provided for illustration. The implementation of dynamic strings is used as an example application of the general-purpose routine.

General Principles

A prerequisite for dynamic memory allocation is a pool of contiguous available memory from which smaller blocks can be allocated. This free memory space is generally called the heap; dynamically allocated subsets of it commonly are referred to as blocks. The physical location of the heap and the way in which it is isolated from other features of the run-time environment are dependent on the language in use and often, its implementation. These details will not be considered here. (Dynamic allocation of space for local variables, which typically uses the stack rather than a heap, will also not be considered.) Other issues related to dynamic memory allocation, such as the identification of free blocks within the heap, the application interface, and efficiency considerations, are more general. The following discussion will address these topics. Note that although the principles described may apply, for instance, to BASIC's dynamic string handling, they will not necessarily allow you to add new dynamic memory functions to BASIC or some other languages.

The Free List

The heart of any memory allocation routine is a data structure that identifies the location of all free blocks of memory; this is conventionally called the "free list." Typically it takes the form of a singly-linked list in which each node identifies the location of a block of available memory, the size of the block, and the position of the next node in the list. At first glance, allocation of storage space for the free list itself would seem to be a problem. Initially, all free memory would be in one block, requiring only one node, but after a series of allocations and de-allocations, the list may contain any number of nodes. Where and how, then, is the free list stored?

One answer is to store the pointers to free memory in the free memory itself. This sounds a bit like a snake swallowing its own tail, but is actually quite simple and straightforward to implement. A small portion of each free block is used to store the block size and pointer to the next node, as shown in Figure 1. The pointers to the free blocks are therefore implicit—the address of each node is itself the address of a free block. One consequence of this method of storage is that free blocks cannot be smaller than a node of the free list. In Figure 1 the nodes are shown sorted from low to high addresses. This arrangement makes deallocation easier, as shown below, but it is not the only scheme that can be used. Nodes may be sorted by block size, for example, to make allocation simpler.

Other methods may also be used to store the free list. The second example shown below uses a bit map, an approach made possible by the fact that blocks are of a fixed size and the total number of blocks is known.

Types of Dynamic Memory Allocation

There are several important distinctions among memory allocation systems. As mentioned previously, one of these is the issue of implicit versus explicit control—whether language features alone can make use of this resource or whether the programmer can use it too. Although implicit and explicit control of memory allocation are not generally found together, they are not mutually exclusive. The dynamic string package presented here, for example, automatically allocates and de-allocates memory to carry out its functions, but can coexist with user programs making explicit use of the same functions.

When a language has sole access to memory allocation functions it can control all pointers to allocated space, and so need not replace each deallocated block back into the free list as soon as the application program releases it. This technique of delayed reclamation of de-allocated space ("garbage collection") allows programs to run faster as long as there is sufficient free memory in the

heap, at the expense of a relatively lengthy delay for garbage collection whenever the heap becomes exhausted. Lisp is an example of a language that manages memory using garbage collection. The technique is particularly appropriate for programs that require dynamically allocated memory but are expected to ordinarily require less than the total amount of memory available. This article describes only immediate reclamation of de-allocated memory; Knuth and Aho et al. should be consulted regarding strategies for garbage collection.

Another important distinction between memory allocation schemes is related to the need for fixed or variable sized memory blocks. An application that creates and destroys only a single type of uniformly-sized structure may use a different strategy than one that manipulates structures of many different sizes. Implementations satisfying these different needs may vary greatly in complexity and efficiency. Some of these differences are illustrated by the examples described below.

A third important factor is the sequence of allocation and de-allocation requests that will be generated by an application. If de-allocation proceeds in the inverse order of allocation (i.e., like a stack), specialized routines tailored for the purpose may be made much more efficient than general-purpose memory allocation functions. Other patterns of allocation and de-allocation requests can lead to varying fragmentation of the free list; memory allocation routines can also be optimized to cope with a high or low degree of fragmentation.

Application Interface

A simple example of dynamic memory use is a program which sorts or counts values in an input file by constructing a binary tree in memory as the file is read. A new node of the tree would be allocated every time a new item is found in the file. The sorted output can then be written to another file during an in-order traversal of the tree. A simple application such as this needs only to be able to allocate additional memory as needed. Any application much more complicated than this, however, will generally need to de-allocate memory as well. If the program described above is extended to read several files in succession, the tree should be de-allocated before the next file is read, to reduce the risk of running out of memory. This application is still simple enough, however, that performance can be improved by reclaiming the entire heap at once rather than de-allocating the tree node by node.

Application programs generally make use of dynamic memory allocation, therefore, via two routines: one to allocate memory and one to release it. These routines are known to C programmers by the names "malloc" and "free" and to Pascal programmers as "new" and "release." Initialization of the dynamic memory buffer and routines is performed by the standard runtime code for these languages. If you write your own memory allocation routines, you will have to take care of this detail yourself, providing a third (initialization) interface to application software. The initialization routine is responsible for marking the entire contents of the heap as available; it may carry out other tasks also, depending upon the needs of the allocation and de-allocation routines.

LISTING 1

```
Screen 1
    0.  ( Dynamic Memory Allocation -- Screen 1 )
    1.  ( Each block of free space begins with a 4-byte control block.
    2.  The first word contains the address of the next free block
    3.  [or 0 if none] and the second contains the number of bytes in
    4.  the current block [including the control block]. )
    5.
    6.  ( Create pointer to beginning of free space, w/ size=0. )
    7.  2VARIABLE FREELIST     0 0 FREELIST 2!
    8.
    9.  ( Initialize memory pool. )
   10.  : DYNAMIC-MEM   ( start_addr  length --   )
   11.       OVER DUP FREELIST !  ( Save starting addr. )
   12.       0  SWAP !  ( Set null pointer. )
   13.       SWAP 2+ !  ( Save length in 1st control block. )
   14.       ;
   15.

Screen  2
    0.  ( Dynamic Memory Allocation, Screen 2: MALLOC )
    1.    ( Returns pointer to n free bytes, or 0 if there is no space.
    2.      Word before returned address holds size of block.  No free
    3.      blocks of less than 4 bytes are allowed.                 )
    4.  : MALLOC  ( n -- n )
    5.    2+ FREELIST  DUP
    6.    BEGIN
    7.    WHILE  DUP @  2+ @ ( Size )  2 PICK  U<
    8.        IF @ @ DUP ( get new link )
    9.        ELSE   DUP @ 2+ @ ( size )  2 PICK -  4 MAX  DUP 4 =
   10.          IF    DROP  DUP @ DUP @ ROT !
   11.          ELSE  2DUP  SWAP @ 2+ !    SWAP @ +
   12.          THEN    2DUP !  2+  0    ( store size, bump pointer, )
   13.        THEN                        ( and set exit flag )
   14.    REPEAT   SWAP DROP ( dump #bytes )  ;
   15.

Screen  3
    0.  ( Dynamic Memory screen 3:  FREE  )
    1.    ( Deallocates memory.  Pointer passed must be from MALLOC )
    2.  : FREE   ( ptr --  )
    3.      2- DUP @  SWAP   2DUP 2+ !   FREELIST DUP
    4.      BEGIN   DUP   3 PICK U<   AND
    5.      WHILE   @ DUP @
    6.      REPEAT                          ( at exit: ( size block ptr1 )
    7.      DUP @  DUP 3 PICK !  ?DUP  ( sz blk ptr1 0 -or- ptr2 ptr2 )
    8.      IF  DUP 3 PICK 5 PICK +  =       ( size blk ptr1 ptr2 t/f )
    9.         IF  DUP  2+ @  4 PICK +  3 PICK 2+ !  @ 2 PICK !
   10.         ELSE DROP THEN                              ( sz blk ptr1 )
   11.      THEN                                           ( sz blk ptr1 )
   12.      DUP 2+ @ OVER +  2 PICK =                 ( sz blk ptr1 t/f )
   13.      IF  OVER 2+ @  OVER 2+ DUP @  ROT +  SWAP !  SWAP @ SWAP !
   14.      ELSE  !
   15.      THEN  DROP  ;
```

Efficiency

The efficiency of dynamic memory allocation is principally a function of the time required to grab and release a chunk of memory. The amount of overhead space (i.e., the number of extra bits required for each allocated block) is also an efficiency consideration, but one that is likely to be less important than that of time. Factors that can affect the time required to allocate or free a block of memory are:

- The amount of free space available.
- The pattern of previous allocation and de-allocation requests; that is, the degree of fragmentation of the free space.
- The size of the block or blocks to be allocated.
- The algorithms used.

Clearly, these all interact in ways that may differ from one application to another and even from one data set to another. If you are concerned about efficiency, your best approach is to evaluate the first three factors as best you can and use them to select appropriate algorithms. Generally applicable analyses of these interactions are probably not possible, although the individual factors may be examined (see Knuth, for example, for a discussion of the effect of memory fragmentation).

Choice of an appropriate algorithm can greatly affect the efficiency of an application. The two techniques presented here provide an illustrative contrast. The general-purpose routine requires two bytes of overhead per block, and the time required to allocate or de-allocate a block depends upon the pattern of

LISTING 2

```
Screen  1
    0.  ( ASCIIZ string manipulation routines  )
    1.  : TEXT ( c --  ) ( Parse text to matching char, put in PAD )
    2.     >IN @ TIB @ + C@ OVER = IF DROP 0 PAD C! 1 >IN +! PAD
    3.     ELSE WORD THEN COUNT DUP PAD + 0 SWAP C!  PAD SWAP CMOVE ;
    4.
    5.  : SCAN0 ( s -- z ) ( Returns address of terminating null. )
    6.     BEGIN DUP C@ WHILE 1+ REPEAT  ;
    7.
    8.  : STRLEN ( s -- n ) ( Return length of string in bytes )
    9.     DUP SCAN0  SWAP  -  ;
   10.
   11.  : CHARS  ( n --  ) ( Define a string buffer of n chars. )
   12.     CREATE 0 C, ALLOT DOES>  ;
   13.
   14.  : STRCPY  ( s1 s2 -- ) ( Copies from s1 to s2 )
   15.     OVER STRLEN 2DUP + 0 SWAP C! CMOVE ;

Screen  2
    0.  ( ASCIIZ string extensions  )
    1.
    2.  ( Return the address of a string literal compiled into
    3.    the dictionary.  )
    4.
    5.  : ('')   ( -- s )
    6.    R> DUP  BEGIN DUP C@ WHILE 1+ REPEAT  1+  >R ;
    7.
    8.  : ''     ( -- s )   ( Example:  '' This string.''  State-smart. )
    9.    34 TEXT  PAD  STATE @ IF   COMPILE ('')
   10.                            DUP STRLEN 1+ HERE SWAP ALLOT STRCPY
   11.                          THEN  ;    IMMEDIATE
   12.
   13.  : PRINT ( s -- ) ( Print the ASCIIZ string at the addr. )
   14.     BEGIN DUP C@ DUP WHILE EMIT 1+ REPEAT 2DROP ;
   15.

Screen  3
    0.  ( More char and ASCIIZ string extensions )
    1.  : UCASE  ( c -- c )  ( Uppercases character. )
    2.     DUP 96 >  OVER 123 <  AND  IF 223 AND THEN  ;
    3.
    4.  : CFROM ( a1 a2 -- a1 a2 c ) ( Gets char from pointer under.)
    5.     OVER C@ ;
    6.  : CFROM+ ( Like CFROM, but increments pointer )
    7.     CFROM  ROT 1+ -ROT  ;
    8.  : CTO   ( a1 a2 c -- a1 a2 ) ( Puts char at top pointer.)
    9.     OVER  C!  ;
   10.  : CTO+  ( Like CTO, but increments pointer. )
   11.     CTO  1+ ;
   12.  : CTRANS+  ( a1 a2 -- a1+1 a2+1 ) ( Transfers a char. )
   13.     CFROM+  CTO+ ;
   14.
   15.  : EOS?    ( a1 -- f )   C@  NOT  ;

Screen  4
    0.  ( More character and ASCIIZ string extensions. )
    1.  : C@C=  ( c addr -- f )    C@ =  ;
    2.
    3.  : STRPOS  ( c zstr -- n )    ( Returns position of c in zstr, )
    4.     0 >R  BEGIN  2DUP C@C= NOT  ( 0-based, or -1 if not found. )
    5.     OVER EOS? NOT AND  WHILE 1+ R> 1+ >R  REPEAT
    6.     C@C= IF  R>  ELSE  R> DROP -1  THEN ;
    7.
    8.  : INSTR  ( c zstr -- f )  ( T if c in zstr, F otherwise )
    9.     STRPOS -1 = NOT ;
   10.
   11.  : STRCAT  ( zstr1 zstr2 --  )  ( appends zstr1 to zstr2 )
   12.     SCAN0  STRCPY  ;
   13.
   14.  : TOUPPER  ( zstr -- )   BEGIN DUP EOS? NOT WHILE
   15.     DUP C@ UCASE OVER C!  1+ REPEAT DROP  ;
```

previous requests. The specialized routine for fixed-size blocks requires only one bit of overhead per block (approximately), in many cases requires near-constant (and minimum) time to allocate a block, and constant time to de-allocate a block.

General-Purpose Memory Allocation

The most important feature of a general-purpose memory allocation scheme is the flexibility to satisfy an indeterminate number of requests for blocks of varying sizes. The most appropriate structure for maintaining the free list under these conditions is a linked list. Each node of the list identifies the position of a free block, its size, and the location of the next block in the list. Generally, this linked list is stored within the free space itself, as shown in Figure 1. The address of each node therefore identifies the position of the associated free block, and this information need not be explicitly stored.

For the sake of efficiency during de-allocation, the free list is generally kept sorted in order of increasing addresses. By using a doubly-linked list, it is possible to make de-allocation slightly more efficient yet (the typical de-allocation strategy is discussed below).

Because each allocated block may be of a different size, and because de-allocation routines are typically passed only the address of an allocated block, the size of each block must be stored when it is allocated. (Modula-2, however, requires the size of the block to be passed to the standard deallocation routine.) It seems that the extra space needed to store the size could be eliminated if the de-allocation routine were passed the size as well as the address, but, as discussed below, in some cases more space is actually allocated than is requested, unknown to the calling routine. For this reason, it is important to store the amount of space actually allocated rather than that requested.

Fitting Strategies

When searching for a free block to satisfy an allocation request, the memory allocation routine can select either:

- the first free block that is large enough (first fit) or
- the block that is closest in size to that needed (best fit).

The first-fit strategy is generally regarded as superior, as the number of small blocks tends to proliferate when using the best-fit method. In addition, because it usually must examine more (often all) of the free list for each allocation request, the best-fit method is slower.

If allocation requests fall into a known pattern, however, you may find that the best-fit method, or some variant of it, is more memory-efficient. For example, suppose that your application most often requests blocks of 30, 50, or 70 bytes. After some period of use, most of the free blocks are likely to also be of these sizes. In such a case, your best strategy may be to choose the first free block of appropriate size, reducing the number of useless 20-byte (approximately) free blocks created.

Eliminating Small Blocks

Wasted space is created whenever a free block is created that is smaller than the application is likely to request. The existence of too-small free blocks slows down the memory allocation routines, as their nodes must be examined each time the free list is traversed. Although it is not always possible to prevent this wastage of space, it is possible to eliminate its effect on performance. This is done by including the "extra" space with the allocated block that would otherwise have left the bytes behind. The actual size of the allocated block, including the "extra" bytes, must be recorded in its reference cell, and the troublesome node can then be eliminated.

An Example of General-Purpose Memory Allocation

An implementation of a general-purpose memory allocation scheme is shown in Listing 1. The example is shown in Forth. Forth encourages the construction of application-specific languages of ar-

bitrarily high level, yet is unsurpassed for the direct memory manipulation needed to implement system routines. In keeping with the Forth philosophy of providing simple tools to build custom applications, there are no standard Forth words for dynamic memory allocation. The examples in these listings are presented in the same spirit: although they are fully functional, they should be regarded as examples only. You should modify, improve, or replace them as appropriate to the needs of your own applications. Heed the dictum about not reinventing the wheel, but be advised to trade in your standard steel-belted radials for racing slicks when the competition gets hot.

The two principal interface words, MALLOC and FREE, are shown in screens 2 and 3 of Listing 1. These routines have the same calling conventions, as well as the same names, as their C counterparts, so even if you know nothing but C, you should be able to make some sense of the Forth code. (Some of the more avid proponents of other languages would say that if you know nothing but C, you know nothing at all; that's a rather harsh judgment, but I would agree that users of languages of the PL-1 family [C, Pascal, Modula-2, and Ada] could profitably broaden their horizons by learning something different: Forth, Lisp, Prolog, APL, and Smalltalk all embody unusual approaches to computing.) This code is written for a 16-bit Forth-83 standard system.

The free list in this implementation is a singly-linked list in which each node occupies four bytes. Each node contains a link to the next, followed by the size of the block in bytes. No free blocks smaller than four bytes are allowed. If satisfying a request from an available block would leave fewer than four bytes, the extra bytes are included in the block being allocated. Except for this limitation, there is no minimum size imposed on either the allocated or free blocks. Free blocks are selected by the first-fit strategy.

The word DYNAMIC-MEM, in screen 1, is used to initialize the heap. It should be passed the starting address and size of the heap in bytes. The heap itself may either be compiled directly into the Forth dictionary or placed in free memory above the dictionary. (If you choose the latter course, take care to avoid conflicts with PAD, TIB, block buffers, and the parameter and return stacks.)

DYNAMIC-MEM creates a single node or control block at the beginning of the heap space, setting its size to be that of the entire heap. The address of this first node is stored in the double variable FREELIST, which has the same format as a node but, having a fixed address, serves as the root, always pointing to the first real node in the free list. The size cell of FREELIST is always zero; it exists so that FREE does not have to treat the root node as a special case.

Each block of allocated memory is preceded by a cell containing the block's size. This information is needed to de-allocate the block. Each allocated block is therefore actually two bytes larger than its nominal size. This overhead cost should be considered if you wish to use the smallest possible heap, based upon your knowledge of the number and size of blocks needed.

The word MALLOC is used to reserve a block; it is passed the number of bytes desired and returns the address of an appropriately sized block, or zero if the request cannot be satisfied. The first thing this word does is increase the requested size by two bytes to allow for the size cell. A sequential search of the free list is then performed, which is terminated when a block of sufficient size is found or the end of the free list is reached. Either of these conditions is signaled by a zero on the stack; the test for this value occurs at the beginning of line 7. During this search two values are kept on the stack: the number of bytes needed and the address of the node that contains the address of the node currently being examined. The address of the node "one back" must be maintained so that that node's link address can be adjusted in case the current node is entirely allocated and must be dropped from the free list.

Line 7 of Screen 2 fetches the size of the current block and tests it against the request. Line 8 performs two fetches to get the link to the next block if the

LISTING 3

```
Screen  1
    0.  ( Dynamic strings, screen 1.  DYNAMEM package must be loaded.)
    1.  : STRVAR   ( Create pointer to dynamic string. )
    2.      CREATE  0 , ;    ( a VARIABLE by another name )
    3.
    4.  STRVAR __SYSSTR      ( Save ptr to created/modified strings. )
    5.
    6.  : LEN   ( dstr --  )    @  STRLEN ;
    7.
    8.  : RELEASE  ( dstr -- )  DUP @  ?DUP IF FREE THEN  0 SWAP ! ;
    9.
   10.  : STRSAVE  ( zstr dstr -- )     ( Assigns zstr to dstr )
   11.     SWAP DUP STRLEN 1+ MALLOC  ( dstr zstr mem )
   12.     SWAP OVER STRCPY   SWAP DUP RELEASE    !   ;
   13.
   14.  : S!  ( dstr1 dstr2 --  )  ( Stores 1 in 2, making a copy )
   15.     SWAP @ SWAP STRSAVE  ;

Screen  2
    0.  ( Dynamic strings, screen 2          )
    1.
    2.  : LEFT  ( dstr1 n -- dstr2 )   ( Returns left n chars of dstr1)
    3.     OVER LEN OVER MIN 1+ MALLOC   DUP >R   ROT @ SWAP ROT
    4.     ( zstr mem n -- )  2DUP + 0 SWAP !   CMOVE
    5.      __SYSSTR RELEASE   R> __SYSSTR !   __SYSSTR   ;
    6.
    7.  : RIGHT  ( dstr1 n -- dstr2 ) ( Returns right n chars of dstr1)
    8.     OVER LEN SWAP - 0 MAX  SWAP @ + __SYSSTR STRSAVE  __SYSSTR ;
    9.
   10.  : SUBSTR  ( dstr1 n1 n2 -- dstr2 )
   11.       ( Substring of dstr1 starting at char n1, of length n2 )
   12.     ROT @  ROT 1- OVER STRLEN MIN +  __SYSSTR STRSAVE
   13.     __SYSSTR SWAP LEFT   ;
   14.
   15.

Screen  3
    0.  (  Dynamic strings, screen 3.     S+  SAY  UPPER  )
    1.
    2.  : S+  ( dstr1 dstr2 -- dstr3 )   ( Appends 2 to 1 )
    3.     OVER LEN  OVER LEN  +  1+ MALLOC  DUP >R    ROT @ OVER
    4.     STRCPY   SWAP @ SWAP STRCAT  __SYSSTR RELEASE  R> __SYSSTR !
    5.     __SYSSTR   ;
    6.
    7.  : SAY   ( dstr --  )
    8.     @ PRINT ;
    9.
   10.  : UPPER    ( dstr1 -- dstr2 )  ( Makes an uppercased copy )
   11.       __SYSSTR S!  __SYSSTR @ TOUPPER  __SYSSTR   ;
   12.
   13.
   14.
   15.

Screen  4
    0.  (  Dynamic strings, screen 4.       S''   )
    1.
    2.  STRVAR __SYSSTR2
    3.
    4.  : (S'')             ( For pre-incrementing NEXTs )
    5.     R> DUP  BEGIN DUP C@ WHILE 1+ REPEAT  1+  >R  __SYSSTR2
    6.     STRSAVE  __SYSSTR2   ;
    7.
    8.  : S''  ( --  dstr )     ( Accepts text from input stream  )
    9.                         ( into anonymous dynamic string.  )
   10.      34 TEXT  PAD      STATE @ IF   COMPILE (S'')
   11.        DUP STRLEN 1+ HERE SWAP ALLOT STRCPY
   12.      ELSE
   13.          __SYSSTR2 STRSAVE    __SYSSTR2
   14.      THEN   ;     IMMEDIATE
   15.
```

LISTING 4

```
Screen  1
    0.  ( Dynamic mem. alloc. for fixed node size, screen 1.       )
    1.
    2.  VARIABLE NODESIZE    ( Size of each node )
    3.  VARIABLE NODEMAP     ( Pointer to bit map of nodes )
    4.  VARIABLE #NODES      ( Number of nodes in heap )
    5.  VARIABLE NODEBUF     ( Pointer to memory buffer )
    6.  VARIABLE SRCHPTR     ( Node # at which to start search for free)
    7.
    8.  : >MASK   ( -1<n<8 --  mask  )
    9.     1+  DUP 2 > IF 1 SWAP 1- 0 DO 2* LOOP THEN ;
   10.
   11.  : NODE   ( n -- m a )  ( n=node #, m=mask, a=address )
   12.     8 /MOD  NODEMAP @ +  SWAP >MASK  SWAP  ;
   13.
   14.
   15.

Screen  2
    0.  ( Dynamic mem. alloc. for fixed size nodes, screen 2.    )
    1.
    2.  : >BYTES  ( n -- n2 )  ( Converts bits to bytes. )
    3.     8 /MOD SWAP 0= NOT ABS +   ;
    4.                                          HEX  .
    5.  : CLEARNODES   ( -- )
    6.     #NODES @  >BYTES  0 DO  FF  NODEMAP @ I + C! LOOP
    7.     0 SRCHPTR ! ;                        DECIMAL
    8.
    9.  : NODEBUFSIZ  ( n1 n2 n3 --  )   ( n1 = address of buffer )
   10.     DUP NODESIZE !                  ( n2 = size of buffer, b )
   11.     1+ /  DUP #NODES !              ( n3 = size of node, b   )
   12.     >BYTES   OVER + NODEBUF !
   13.     NODEMAP !
   14.     CLEARNODES    ;
   15.

Screen  3
    0.  ( Dynamic mem. alloc. for fixed size nodes, screen 3.  )
    1.                                        HEX
    2.  : GETNODE   ( -- a )   ( a = 0 if no space available )
    3.     0 ( accumulator )  #NODES @  0 DO  I SRCHPTR @ +
    4.     #NODES @ MOD  DUP   NODE C@ SWAP AND ( free? )
    5.     IF     DUP 1+ #NODES @ MOD SRCHPTR !
    6.            DUP  NODE  DUP C@  ROT  FF XOR  AND SWAP C!
    7.            SWAP DROP  NODESIZE @ *  NODEBUF @ +   LEAVE
    8.     ELSE  DROP
    9.     THEN  LOOP  ;
   10.
   11.  : RELEASENODE   ( a -- )  ( a  as returned by GETNODE )
   12.     NODEBUF @ -   NODESIZE @ /
   13.     DUP SRCHPTR !
   14.     NODE DUP C@  ROT OR  SWAP C!  ;
   15.                                        DECIMAL
```

size is insufficient. Line 9 evaluates whether the entire block should be allocated; if so, the pointers are adjusted in line 10, otherwise the size of the current block is reduced in line 11. In either case, the address of the block is left on the stack. Line 12 stores the size for later use, increments the pointer past the size cell, and sets a zero flag on the stack to terminate the loop.

Release of an allocated block may or may not result in the addition of another node to the free list. Blocks above and below the one to be de-allocated may themselves be either free or reserved. The four possibilities are shown in Figure 2. Only when the memory configuration is as shown in Figure 2a will a new node be added to the free list. The situation shown in Figure 2b will result in the creation of a new node within the newly de-allocated block, and the removal of the node above, for no net change. The link address previously pointing to the node to be removed must also be modified. When the situation is as shown in Figure 2c, only the size of an existing node need be changed. If free memory bounds the de-allocated block on both sides, as in Figure 2d, then the size of the lower node must be changed and the upper one eliminated.

The need to examine the blocks on both sides of the one to be de-allocated is why the free list is kept sorted by address. To find the address of the preceding free node, a sequential search is performed for a node which has an address lower than that of the one to be de-allocated, but a link address that is higher. If the size and address of the lower node sum to the address of the one to be de-allocated, then the situation in either Figure 2c or 2d applies. To find the address of the following block (which will have a free-list node if empty), it is only necessary to sum the size and address of the block to be de-allocated; if the resulting address appears in the free list, then the situation in either Figure 2b or 2d applies.

Evaluation of the memory configuration and removal of the indicated node are performed by the word FREE in Listing 1, Screen 3. This word begins by fetching the size of the node and storing it in the second cell, creating the size cell of a valid node header. A sequential search of the free list is then performed (lines 3-6), ending with the address of the free node below the one to be de-allocated. Note that this may be the root (FREELIST) which, because of the extra cell allocated to it, may be treated exactly like any other node header.

In line 9, the link address held by the next-lower free node is stored in the block to be de-allocated, completing the valid node header for this block. Nothing yet points to this header, and it may eventually be abandoned. Construction of the header at this step is more efficient, however, if the node is not to be abandoned. Lines 8-10 evaluate whether the node to be de-allocated is immediately followed by a free node; if so, the size cell of the newly created node header is increased by the size of the following free block and the link address is set to that contained in the following header. Lines 12-14 evaluate whether the block to be de-allocated is preceded by a free block; if so, the link and size cells of the preceding header are modified appropriately, and if not, the link address of the preceding header is set to that of the de-allocated block.

An Example Application

The use of these words is illustrated by a set of routines for manipulation of dynamic strings. Listing 2 contains a set of static string-handling words, and Listing 3 ties these together with the dynamic memory words in Listing 1.

Strings are generally stored in memory in one of two ways: with the string length in the first byte or word, or with the end of the string marked with a sentinel character, usually an ASCII zero. For simplicity, I will refer to these alternatives as "counted strings" and "zstrings". Dynamic strings will be referred to henceforth as "dstrings". Forth contains several standard words for manipulating counted strings (using a single byte for a count), but is not limited to this form of storage. I prefer to use zstrings, as they allow you to scan a string more easily; the remainder of the string can always be represented by a single stack element rather than by an address-count pair, as is necessary with counted strings. The words in Listing 2 are therefore designed to create and manipulate zstrings rather than the more usual (for Forth) counted strings.

Because this is an illustration and not central to the point of this article, the words in Listing 2 will not be described in detail. A few points are worth noting, however. In particular, the words TEXT and (") may be found in existing Forth systems with slightly different actions. Typically these create and return counted strings, whereas the versions shown here are designed for zstrings. If possible, you should rewrite SCAN0 in your native assembly language, as it may amount to only a single instruction. The words in Listing 2 do not form a complete set of tools for handling static strings, but they include all those used to illustrate dynamic string handling in Listing 3 as well as a few others.

The words in Listing 3 integrate those in Listings 1 and 2. They allow strings of any length (within the limits of the heap space) to be stored or modified

without any concern on your part about overrunning a statically allocated string buffer. These words mimic some of the string-manipulation functions of dBase, in name and application.

Dynamic strings are represented by a pointer to a zstring; the zstring itself is stored in the heap rather than in the Forth dictionary. A dstring can be converted to a zstring simply by a fetch (@) operation. With that in mind, and an explanation of the role of __SYSSTR, the words in Listing 3 should be easy to interpret.

Several of the dynamic string manipulation routines create new unnamed dstrings—that is, ones that do not directly replace one of the dstrings passed as a parameter. The words LEFT, RIGHT, and S+ are examples. This new, unnamed, dstring is left on the stack, where you may save it (with S!), display it (with SAY), or otherwise dispose of it. The pointer to the heap space allocated for this string must not be lost, however, or the space will be unrecoverable. __SYSSTR is used to store this pointer. Note that the pointer is stored only until the next operation that creates a new unnamed dstring; at that point the space is de-allocated and the pointer reassigned. In some situations, this limits the operations that can be successively carried out on an unnamed dstring. Consider the following sequence of commands:

```
STRVAR COMPOST

" Gardeners rarely grow cabbage."  COMPOST STRSAVE

COMPOST 3 LEFT

COMPOST 5 RIGHT

S+
```

The result of this would be garbage, but not the "Garbage." that you might expect. Both of the phrases COMPOST 3 LEFT and COMPOST 5 RIGHT leave a pointer to an anonymous zstring, but only one anonymous pointer (__SYSSTR) is allowed. Thus, the two arguments passed to S+ will both be __SYSSTR, and the result will always be to concatenate the rightmost five-character substring of COMPOST with itself. The solution to this problem is to use another dstring defined with STRVAR for intermediate storage of the leftmost substring.

Any number of successive operations on a single unnamed dstring may be carried out, however. For example:

```
COMPOST 6 LEFT  UPPER  SAY
```

These routines are written so that __SYSSTR may be one of their arguments, and space for the resulting string will be allocated before __SYSSTR is de-allocated.

Another way of reducing conflicts between uses of __SYSSTR is to use a different system string for each routine. This approach is taken with the word S" (the dstring counterpart to "), simply to allow the convenience of entering a string while an unnamed dstring resides on the stack. The drawback is that heap space may remain allocated long after the unnamed dstring is no longer needed by the application.

The technique of implementing dynamic strings shown in Listing 3 is only an example. Counted strings could be used instead of zstrings. The count could also be kept in the dstring header, whether counted strings or zstrings are used. This last approach may be most suitable when you want to use zstrings for most purposes, but your application frequently needs to evaluate the length of strings; the extra space devoted to storage of the string size, although unnecessary, may save computation time. Tailor the tools to the task.

Special-Purpose Memory Allocation

If there is anything systematic about the size of blocks that will be needed, the number of allocation requests, or the pattern of allocation and de-allocation, you may be able to improve performance and save memory by using a special-purpose memory allocation routine. Whereas most general-purpose memory allocation routines will probably be based on a model somewhat like that presented above, you are pretty much on your own when it comes to designing a special-purpose routine. Knuth and Aho, Hopcroft, and Ullman describe a technique known as the "buddy system," which is a sort of general-purpose special-purpose system, suitable when only a limited number of sizes of blocks will be needed. Its advantage is that it can be customized for different combinations of sizes of blocks.

Considerations of fitting strategies and the problems of small blocks do not pertain when all blocks are the same size. It is, in fact, easier to design an appropriate solution for a single special-purpose application than it is to design a good general-purpose memory allocation routine.

The technique described here is one that is suitable only when blocks of a single size will be needed. But for this limitation, it has a number of advantages over the general-purpose routine described above:

- The time required to allocate a node is likely to be much less.
- The time required to de-allocate a node is constant.
- The overhead is only one bit per block rather than two bytes.

These advantages are conferred by the representation of the free list as a bit map rather than as an actual linked list. The bit map consists of a series of bytes long enough so that their total number of bits is at least as great as the number of nodes that can be accommodated by the heap. The state of each bit (set or reset) indicates the availability of a corresponding node. The code for this implementation is shown in Listing 4. The words NODEBUFSIZ, GETNODE, and RELEASENODE are analogues to DYNAMIC-MEM, MALLOC, and FREE in Listing 1.

A free block is indicated by a set (1) bit in the map. To allocate a block, it is necessary to scan the map for such a bit and calculate the address to which it corresponds. To increase efficiency when a sequence of successive allocation requests may be performed, each search of the map begins where the previous one left off. To increase efficiency when an alternating sequence of allocation and de-allocation requests is performed, a pointer is set whenever a block is de-allocated so that the next search will begin with that block and so will be satisfied immediately. In some cases only one of these fine-tuning mechanisms may be appropriate; both are shown here for illustration.

The housekeeping information is kept in the five variables shown in Listing 4, Screen 1. The first three words (>MASK, NODE, and >BYTES) manage the conversion between the bit map and actual addresses. The word >MASK ("to-mask") takes a bit number and converts it into a mask that can be used to test or set the bit with AND or OR. This is a good candidate for coding in assembly language.

Initialization of the bit map and housekeeping information is performed by the word NODEBUFSIZ. The beginning of the memory buffer is set aside for the bit map; this routine calculates the number of nodes that will fit and the size of the map needed. The map always occupies an integral number of bytes. Depending upon the buffer and node sizes, up to seven bits of the last byte of the map may be unused. The overhead per block may therefore be slightly more than one bit.

The word CLEARNODES has been factored out of NODEBUFSIZ so that it can be used to re-initialize the buffer without the need to use RELEASENODE to de-allocate each block. This word must be used with great care and subsequent reference to dangling pointers should be avoided.

GETNODE (Listing 4, Screen 3) allocates space by looping over the total number of nodes; the phrase

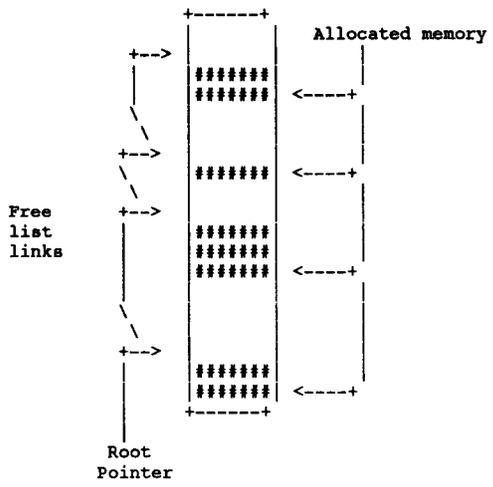


Figure 1. Linked List in the Heap

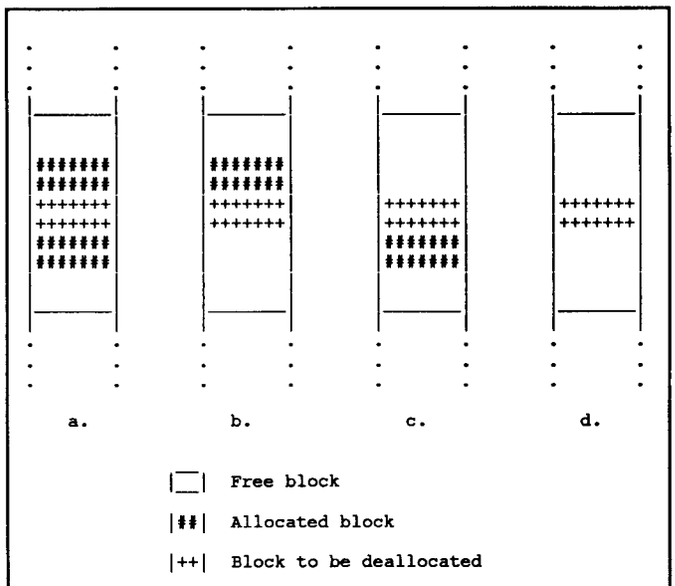


Figure 2. De-allocating a block; surrounding memory

```
I SRCHPTR @ +  #NODES @ MOD
```

translates a relative node number into an actual node number based upon a non-zero starting position. If a free block is found, the starting point for the next search is set (line 5), that entry in the map is marked as allocated (line 6), and the actual address of the block calculated (line 7).

The word RELEASENODE de-allocates space by calculating the node number (Screen 3, line 12) and clearing the appropriate bit (line 14). In addition, it sets the starting point of the next search to the node just de-allocated (line 13).

Because of the uniform block size, this approach lends itself to compressed displays of the allocation map more easily than does the first. A simple word to display this map may be defined as follows:

```
: SHOWMAP
    #NODES @ 0 DO
       I NODE C@ AND  IF
          ." 1"
       ELSE
          ." -"
       THEN
    LOOP  ;
```

Summary

The examples shown in this article, although useful in their own right, are intended principally to illustrate a point. That is: you can improve the performance of your application programs by tailoring memory allocation routines to their specific needs.

Several changes could be made in the general-purpose routine that might improve its suitability for certain applications. For example, each search could be started wherever the previous one terminated, as is done with the specialized routine. Also, backward links in each node header would eliminate the need for a sequential search when a block is to be de-allocated.

If the size of blocks is known at compile time (which is very often the case), the special-purpose routine could be improved by making NODESIZE a CONSTANT rather than a VARIABLE. Depending upon the actual block size (e.g., for powers of two), other changes may also increase performance. See the references.

Other special cases of memory allocation, such as a series of LIFO requests, may be handled by techniques very different from either of the examples shown here.

Although the standard libraries of most conventional languages provide routines only for general-purpose memory allocation, you can still take advantage of opportunities to create special-purpose routines as needed. If you cannot supplant the standard routines, they can at least be used to permanently allocate a heap large enough for your own routines to use. For some applications you may even wish to have two or more different memory allocation techniques in use simultaneously, each with its own heap. Consider the needs of your application carefully, use the techniques shown here and in the references as guides, and you can design memory allocation routines that provide optimum performance. ●

REFERENCES

Aho, A.V., J.E. Hopcroft, and J.D. Ullman. 1983. Data Structures and Algorithms. Addison-Wesley, Reading, Mass. 427 pp.

Knuth, Donald E. 1973. The Art of Computer Programming. Volume 1, Fundamental Algorithms. Second Edition. Addison-Wesley, Reading, Mass. 634 pp.

Using BYE with NZCOM

The New Taming of an Old Shrew

by Chris McEwen, Sysop Socrates Z-Node #32

Socrates, my rcpm, went on line last December. Evidently, this was more of an event than it seemed at the time. Why? I had just bought NZCOM the week before, without any previous Z System experience, and getting BYE to peacefully coexist with NZCOM was supposed to be hard. To be fair, mine wasn't the first rcpm to run under NZCOM. Bob Dean converted Drexel Hill to NZCOM sometime the previous summer, and I am sure there were others. But the difference, I'm told, was that a total neophyte managed to stumble in the right combinations to make things work. This seemed to interest Jay Sage, who surely is more accustomed to dealing with people who can walk and chew gum at the same time! He asked me to tell you how I did it.

Before we start, I should mention one thing. It is true that you can't run NZCOM under BYE. BYE is an RSX and protects itself from being overwritten. NZCOM is a very powerful loader that can reconfigure the memory map. It looks for such programs as BYE and refuses to run when they exist. But we don't need to run NZCOM while BYE is running. We run it **before** we run BYE, and we change systems with ENV files rather than ZCM, using JetLDR. Our only real restriction is that we cannot change the memory map while BYE is active.

In this article, we will set up a Xerox 16/8 DEM-II with a 10 meg hard drive, which we have configured with three logical drives (A: through C:) for the hard drive and one floppy as drive D:. Figure 1 lists the steps to take, which we will discuss in turn.

```
            Get NZCOM running first.
              RCP vs. Transient commands.
              Become familiar with ZCM files and how to edit them.
              Make your named directory files.
            Patch WHL.COM and NZCOM.COM
            Get BYE running next.
              Watch out! There are some traps here.
              Z3BASE.LIB
    +-+-+-> Tweak it.
    | | |     Use MKZCM, NZCOM.COM and JETLDR.
    | | |     Current public DU:'s will reflect in the new .ZCM files.
    | | +-< Check SHOW, PATH and PUBLIC, and the BYE.PRN file.
    | |     Get your BBS software up and running.
    | +---< Make your aliases.
    |         My usual ones.
    |       Choose your transient commands carefully
    |         What stays on A0:
    |         What must be moved to A15:
    +-----< Check the system on line.
              Watch for security.
```

Figure 1: Steps needed to run NZCOM and BYE.

Get NZCOM Running First

Why NZCOM first? It is your operating system. Imagine trying to run a program without having CP/M installed on your computer. BYE is a nasty program in that it hooks itself very deeply into the system. Getting it running under the wrong system is a waste of time.

We want to get the memory configuration of NZCOM that you will use with the BBS going. If you need a certain sized TPA for your BBS, you have to make room for it here since we can't change the memory map later while BYE is running. Place MKZCM, SHOW, PATH, PUBLIC and your favorite editor on A0:. Run MKZCM to create your 'on line system'. We will be making several versions of the system, but they must all have the same memory map.

We will be setting up three systems. The first is the one we will let the callers use. It will have significant restrictions set on it. Then we will set up a system for the sysop which will allow you to do anything you like on your computer, but will lock out the floppy disk drive. Why do that? What if you call in remotely and enter a command such as 'FF' that accesses the floppy, but you forgot to leave a disk in the drive? You'd hang the system. Finally, we'll make one last system the same as the sysop's, but it will let you at the floppy. I found it easier to set up the restricted system first and then after that is running properly, go back and set up the sysop systems.

I installed an NZCOM system without any RCP. As I implied in the lead paragraph, my assembly programming experience is less than minimal. As a result, I trust transient commands much more than I do anything permanent in the operating system itself. If a command doesn't behave as I expected, I replace it, or get it out of harm's way. The book says that IOP's are a topic for advanced users. Well, that did that! I dumped them as well. I then increased the number of named directories allowed. And with that, I saved my new system. Use the name 'USER' to save this configuration.

MKZCM will save two files, each of which describes the configuration you've just done. USER.ZCM is of particular interest to us as it describes the target system in a text file which you can easily edit. Let's do that now. Pay particular attention to MAXDRV, MAXUSR, QUIET, Z3WHL, DRVEC, PUBDRV, and

About the author and his system:

Chris McEwen is a management analyst living in central New Jersey. He has been running public bulletin boards since 1985 but only established a CP/M-based system at the beginning of 1989. Within three months, Socrates had gained Z-Node status. Chris dedicated Socrates to learning, whether it be the Z-System or high level languages. There is a message base devoted to the new 'C' programmer. In addition, Socrates is the central site for QBBS development.

Socrates can be called at (201) 754-9067, at up to 2400 bps. It runs on an Ampro Little Board with a 64 meg drive. Chris runs on Coke and potato chips.

```
   EA06 CBIOS    0080 ENVTYP    E8F4 EXPATH    0005 EXPATHS   0000 RCP
   0000 RCPS     0000 IOP       0000 IOPS      E200 FCP       0005 FCPS
   E480 Z3NDIR   0023 Z3NDIRS   E900 Z3CL      00CB Z3CLS     E780 Z3ENV
   0002 Z3ENVS   E700 SHSTK     0004 SHSTKS    0020 SHSIZE    E880 Z3MSG
   E8D0 EXTFCB   E9D0 EXTSTK  ->0000 QUIET    ->E8FF Z3WHL    0004 SPEED
->0010 MAXDRV  ->001F MAXUSR    0001 DUOK      0000 CRT       0000 PRT
   0050 COLS     0018 ROWS      0016 LINS     ->FFFF DRVEC    0000 SPAR1
   0050 PCOL     0042 PROW      003A PLIN      0001 FORM      0000 SPAR2
   0000 SPAR3    0000 SPAR4     0000 SPAR5     CB00 CCP       0010 CCPS
   D300 DOS      001C DOSS      E100 BIO      ->0001 PUBDRV  ->0080 PUBUSR

Figure 2: USER.ZCM
```

PUBUSR. Load up your editor and bring up USER.ZCM in non-document mode (see Figure 2).

This almost describes a Xerox 16/8 DEM-II computer, but it is wrong about the drives we have. Notice that MAXDRV is 0010, and DRVEC is FFFF. These two values say that we have 16 contiguous drives on the computer. This is not the case. This system has four drives, but we are building a system for public use, and we won't be letting the callers at our floppy drive. We need to change MAXDRV to 0003.

That's easy enough. But what of this DRVEC? It is a bit map of the valid drives, which lets NZCOM skip over any drive that is not present. You can use the following chart to determine the value to give DRVEC. Put a one over any drive that you have on the system. Add up the values for each line, and write them down in hexadecimal to the right.

```
Weight Factor:
8    4    2    1

0    0    0    0    =    0---
P    O    N    M

0    0    0    0    =    -0--
L    K    J    I

0    0    0    0    =    --0-
H    G    F    E

0    1    1    1    =    ---7
D    C    B    A
0007
```

Change DRVEC to 0007.

We also want to limit the highest user area we will let the callers have access to. All the sensitive commands such as ERA will be up high. I have mine set at 7. Change MAXUSR to 0007.

The QUIET flag tells the system if it should report what it is doing to the user. We want this for ourselves, but not for our callers. Part of our security is that we will be using aliases to load in modules which will be given secret names. If the quiet flag is off, the names will be reported as they load. So set QUIET to 0001.

Take note of the value you have for Z3WHL. You will want this later on when we get to BYE. Save this file.

But didn't we forget PUBDRV and PUBUSR? These refer to the public drive and user areas that ZRDOS will recognize, and are a bit of a bear. On my system, I have A8: set as a public DU: where I put WordStar. Obviously we don't want callers using that! But every time I edited the USER.ZCM file to say there were no public DU's, the next time I loaded the system, they'd be back! The trick here is to use the PUBLIC utility to cancel any public DU's before you load your new system. Do that now.

Now enter 'NZCOM A0:USER.ZCM' to load this system. Be sure you include the prefix A0:. Run SHOW to see if we have the values we want for the drives and user areas. You'll see this on screen 3. Everything OK? If not, then go back to your editor and change USER.ZCM as needed.

Run PATH to see if the QUIET flag is correct. It won't tell you anything if the QUIET flag is on. If it tells you what your path is, then the QUIET flag is off. That's not good. Again, load your editor, and fix QUIET.

If you've changed anything, reload with NZCOM and check everything again with SHOW and PATH. Keep editing, reloading, and checking until you have it the way you want it.

Now check for PUBLIC DU's. You should have none. If you do have any, clear them now.

Run MKZCM one more time. Don't change anything, just save it under the same name. Why do that? Remember that MKZCM creates two files? The one we've been working with has the extension of 'ZCM'. If you noticed, the other file MKZCM saved had the extension of 'ENV'. This is what we've been after all this time because JetLDR can handle this file just fine.

Check and recheck that the system is set as you'd want for open use. When you are happy with the users' system, we will go on to make the sysop system. Bring up MKZCM again, but this time save the result under a name that only you will know. For our discussion, we will call it SYSOP. Let's go back with your editor and give you some access on your own computer!

Change MAXUSR to the maximum user area you have. This is usually 15. Pull that DRVEC chart out again. Check off all the drives you need access to, **except** for floppy disks. Then set QUIET to 0000. But watch out! Don't do anything that changes the size of the system. Save the results.

Enter 'NZCOM A0:SYSOP.ZCM' to load this system. Again, it is important to enter the A0:. Run SHOW and PATH. Is it set as you want? If not, edit again and reload.

Now set any public DU's you want. After you've thoroughly verified the settings, run MKZCM to create an ENV of this system. Finally, create one more system, but this time include the floppies. Give this another secret name.

What have we done? We've created three environment files that we can use on-line to change a caller's access. We don't need the ZCM files any longer, so you can erase them. Use STAT or DFA to set all the ENV files to $SYS so that users will not be able to see them with the DIR command.

The last thing to do before we move on is to create the named directory files. I use the same names as the environment files. The big point here is that even if a DU: is out of range of the environment, if it has a name and no password, a caller can move there. You can give passwords to directories, but it is simpler just to not declare them in the first place if you don't want people going there.

[Note by Jay Sage: I take a different approach and make extensive use of named directories with passwords. In fact, the named directories on my system are the same for users and sysops. All I do to make the sysop systems is turn on the wheel byte, since when this is on, passwords are ignored, and one has free access to all the sysop directories.]

Patch WHL.COM and NZCOM.COM

Before we go too much further, you need to make two patches. Make backup copies of NZCOM.COM. If you dumped the RCP as I did, you need a transient called WHL32.COM to manipulate your wheel byte, and we will patch this as well. If you are using the RCP, your system password is in there. Big point here is to do this after you've made back-up copies of whatever you are going to patch. Can you say 'oops'?

Use DU (disk utility), ZPATCH, or whatever you are comfortable with and call in NZCOM.COM. Search for NZCPM. This will be in the FCB section of the program. Change it to something else. Your restrictions are that you must make this eight characters or less, that you must pad it out to exactly eight characters with spaces, and that you must use capital letters. What you put here must be a secret.

Now, why did we do this? NZCOM will make a file called NZCPM.COM on the disk if there isn't already one. The purpose of this file is to allow you to dump the

NZCOM system and go into straight CP/M. If a user does this on line, he will effectively turn your BBS off. He can't hurt anything, as BYE won't be able to talk to the system any longer, but it won't reset when he finally drops carrier, either. You'll be crashed until you reboot.

So we gave NZCPM a secret name. Drop out of NZCOM and reload it. The system will write NZCPM.COM under the name you just gave it. Erase NZCPM.COM, and use STAT to make its replacement a $SYS file so that no one but you knows its name.

[Note by Jay Sage: Again, I can suggest an alternative and simpler approach. Leave NZCOM.COM as it is. Run it to create the file NZCPM.COM, and then copy that file to a secure area. Then use SALIAS to create an alias called NZCPM that has the script command: "IF WH;DIR:NZCPM;FI", where DIR is the directory where you put the real NZCPM.COM. The presence of this alias will inhibit NZCOM from creating a new NZCPM file, and the alias will do something only in sysop mode (when the wheel byte is on). If the wheel byte is off, the command will do nothing. If the wheel is on, then the real NZCPM command will be invoked.]

The other patch we have to make is the wheel password. If you dumped the RCP as I suggested, then you will be using WHL32.COM. Patch that. Otherwise you patch NZRCP.ZRL in NZCOM.LBR. Look for either SYSTEM or PASSWORD. I forget what it says in the distribution copy. Change it to something else. Again, your restrictions are eight characters, padded with spaces, in capital letters. [Note added by Jay Sage: This patch you absolutely must do; you must not leave a wheel-setting command on the system with an unsecure password. The wheel password is not determined by the system but is set for each WHEEL program (e.g., WHL32 or the RCP WHL command). You should be able to find the password using a patching utility and change it to something secret. Be sure to test it before putting your system on the air.]

Get BYE Running Next

Now comes some real fun. Getting BYE running for the first time is almost guaranteed to take five years off your life and is more that we can tackle in one article. I suggest you work closely with a Z-Node sysop for assistance as you go. But here is the plan: get BYE running any way you can at first, and then go back to tweak it. I would suggest you rename DIR to the name of the BBS you plan to run so that it will be the program run when you test BYE. This eliminates any problems you may have with your BBS system as you debug BYE itself.

BYE is a necessary evil. It hasn't been given a full rewrite in about five years, and its age is showing. The biggest problem is that it tries to be all things for all systems. All I want from BYE is modem redirection, a few extra BDOS calls to handle situations that would only happen under a remote system (such as time on line and carrier test), and maybe a few neat function keys like "Who's on line?". What I don't want it doing is messing with the environment. We have an operating system to do that for us. Unfortunately, BYE insists, and it usually messes things up. One of these days we will have a BYE made for today's systems. Until then, we have to work with this monster. [Note added by Jay Sage: See my column in TCJ #40 for a discussion of what BYE does. I second Chris' comments about BYE and the need for a replacement that is appropriate for Z-Systems.]

I use QBYE, as it is the simplest to set up. QBYE is based on NUBYE 1.01 by Tom Brady. Tom and Irv Hoff had worked together for most of the development of BYE but parted company just as the last generation came out. I would expect whatever findings I have with QBYE you will have with BYE 5.10.

I noticed some very odd happenings at the OS level and suspected a conflict between BYE and NZCOM. There were two symptoms: the utilities that check the DRVEC seemed to be pretty solid, but those that checked MAXDRV were flaky. For example, FF (Find File) would not report any files found on the highest drive. If I set the system to sysop access while a user was on line, it acted strangely once I would reset back to normal access. The only solution was to allow the caller to have wheel privileges for the duration of the call.

Finally, I pulled SHOW down while a caller was on line to see what was going on. It seems that BYE was resetting the MAXDRV and MAXUSR bytes erroneously. On cold boot, it was giving MAXDRV one less drive than allowed, and MAXUSR one more. More importantly, once any new environments were loaded, it put invalid data into these bytes.

Though I had told BYE not to monitor the maximum DU: settings, it insisted on doing just that. Worse, it wasn't doing it right! See Figure 3 for the CCP settings in the BYE configuration file as used on Socrates. Be aware that ALL system security with these settings is now the purview of NZCOM. BYE will not monitor anything for you. Carefully test your various environment settings remotely before leaving the system for public calls. You should look through the PRN file to make sure the proper addresses are being assigned, since the addresses will differ from system to system.

You will notice reference to an include file named Z3BASE.LIB. You will have to generate such a file with definitions for the module addresses referenced in BYE. Figure 4 shows the Z3BASE.LIB that I use. You have to edit this with your memory configuration before you assemble BYE. Notes in the file will explain.

So now you have BYE running. Go online and use SHOW to make sure the system has stayed the way you want it to. Use JetLDR to load the various environments we made up before and use SHOW to verify that MAXDRV, MAXUSR, and DRVEC have stayed correct. Then, turn your WHL on and off while you try wheel-dependent commands such as ERA. The system should respond correctly. If you have problems, you need to edit either your Z3BASE or BYE again and reassemble.

Once you have gotten this far, you are ready to install your BBS software. I use QBBS for a couple of reasons. It holds messages from different areas completely apart, and it is distributed with full source code. It doesn't hurt that QBBS is almost a snap to install. What is taken as a negative by many, that it is written in compiled BASIC, is a plus in my mind. What does a BBS program do? Basically, it is a text file reader that has to be capable of finding messages quickly. Other than that, and the message editor, a BBS program really isn't that involved. I will put QBBS up against PBBS and HBBS, both written in 100% machine code, in a speed test any day of the week. Also, modifying high level language programs is usually easier. But what you chose is up to you.

Make up Your Aliases

As I said earlier, part of your system security is that the names you give your environment files must be a secret. The only way to invoke them with a caller on line is to blank out the modem output with BYE's ESC-B, or to load them through an alias. I use the alias method. If you haven't picked up on it by now, I don't trust BYE farther than I can throw it....

Here are a couple of example aliases I have. By the way, don't put these into your ALIAS.CMD file. I've seen various versions of TYPE that let users type out a $SYS file, and that would blow the secret!

This is the alias to load the normal (secure) system. It is named NZUSER:

```
A0:NZUSER
ldr a0:user.env
ldr a0:user.ndr
whl <wheel password> /s
path a0 $$$$ a0
whl r
echo system load done
```

Now the alias to load the sysop system:

```
A0:NZSYSOP
if ~wh
whl /s
fi
```

```
if wh
ldr a0:sysop.ndr
ldr a0:sysop.env
path a0 $$$$ a15 A0
echo sysop system loaded
else
echo access denied
fi
```

This alias gives the user a chance to set the wheel in case it is off, but will abort if he can't get it set.

Two questions. First, why do we load the SYSOP.NDR before we load the SYSOP.ENV? Remember the QUIET flag? If we reversed the order, the system would report the name of our NDR file to the user. Second, why do we load the extended path after we load the environment? Because if we didn't, A15: would be an invalid DU:, and the system would refuse to allow a path to it.

The alias to load the floppy system is the same as the sysop alias, except it loads the floppy environment.

The last of what I feel are the essential aliases is called BYE. Why would I do that? Again, I don't trust the real BYE to handle system security properly, so I have this alias reset the environment through the NZUSER before calling the real BYE. Of course, rename your real BYE to something else, and make it a $SYS file:

```
A0:BYE
echo one moment please.
nzuser
echo thank you for calling.
echo please call again.
realbye $*
```

Choose Your Transients

You are very close to going on line. Move MKZCM, SHOW, STAT, your editor, and anything else that allows someone to fool with the system up to a safe, high user area. Most of us use A15: for this. Set all the ENV and NDR files to $SYS status, as well as all NZCOM files and libraries and the aliases we made up. Not only does this keep people from trying things they shouldn't, it also keeps them from downloading them. What good does it do to go through all this to have someone download your NZCOM.LBR with its patched wheel password?

Time to choose your transient commands. You will need something for file transfers. I use ZMD150 and RZMP16. Something to type out text files? I use ZLT12. Something to lock into LBR and ARC files? I have LUX77B, LUSH, and ZLUX26, none of which I am really happy with. Gotta work with ARC files, like it or not, so that means you need UNARC16. Don't forget LDIR, and in today's world, ZIPDIR. Does that about do it?

Let's Go See the World

If you've gotten this far, you're ready to start taking calls. I suggest you start by calling it yourself! Thrash it, bash it, try to break it. If you can't, then it is time to tell a few friends. Give them the same assignment. Have them do anything they can to crash the system. If someone can do it, eventually they will, and it might as well be now, done by a friend who will tell you how it happened. Leave the system private amongst yourselves for a couple of weeks. If it still works as it should after this time, go public. We will all welcome a new RCP/M.

Welcome to the club, sysop! •

```
; ++ CCP Options ++
;
ZCPR2   EQU     no              ; Yes, if running ZCPR/ZCMD/NZCPR (1 or 2)
;
; NOTE: Requires MAC.COM to assemble if ZCPR3 is set YES.
;
ZCPR3   EQU     yes             ; Yes, if running ZCPR3
;
         IF     ZCPR3
        MACLIB  Z3BASE          ; Requires MAC to assemble
         ENDIF
;
; NZCPR/ZCMD/ZCPR all use bytes (at 3DH/3EH/3FH) to store the maximum
; drive, wheel status, and maximum user area.  QBBS pokes these values
; in QBYE which in turn maintains them in low memory bytes.
;
USEZCPR EQU     yes             ; (QBBS = NO, except w/NZCOM. Then, YES)
;                               ;
CHEKDU  EQU     no              ; Yes, if QBYE will monitor MAXDRIV/USER.
                                ;   If using ZCPR/ZCMD/NZCPR, set this NO,
                                ;   since they already do it (saves a lot of
                                ;   code, too).  In either case, QBYE will
                                ;   have the correct values in MAXDRIV/USER.

;Set this equate to your system's ENV address:
NZENV   EQU     0E780h          ; Required for use with NZCOM
                                ; this value will vary on each computer.
                                ; use SHOW to see where your ENV is.
WHEEL   EQU     NZENV+17Fh      ; Location of ZCPR's wheel flag
MAXDRIV EQU     NZENV+02Ch      ; ZCPR location of MAXDRIV byte
MAXUSER EQU     NZENV+02Dh      ; ZCPR location of MAXUSR byte
;
MAXDRV  EQU     'J'             ; Highest drive supported
                                ; NZCOM:  Put this to highest + 1 on system
                                ; and let the OS control access.
MAXUSR  EQU     15              ; Highest user area
                                ; NZCOM:  Put this to highest on system and
                                ; let the OS control access.
;
; In all cases, set SYSDRV/USR, since the ^B function gives you these
; d/u areas when used to toggle off the user temporarily.
;
;NZCOM:  Set SYSDRV to one more than you really want.
è
SYSDRV  EQU     'J'             ;#Highest local drive supported
SYSUSR  EQU     15              ;#Highest local user area (0-15)
;
```

Figure 3. This is a section of the BYE configuration file showing thelproper settings to use on an NZCOM system.

```
;Z3BASE.LIB
;
;Last edited: 10 July 89, Lee McEwen
;
;Currently configured for use with:
;  Ampro LB, 64 MB / NZCOM
;  Maximum memory size for use on bbs under bye
;
false   equ     0
true    equu    not false
off     equ     0
on      equ     not off

base    equ     0

;The following values are taken from screen 1 of SHOW:

z3cl    EQU     0DD00H                  ;mcl, multiple command line
z3cls   EQU     203                     ; length of mcl in bytes
expath  EQU     0DCF4H                  ;path
expaths EQU     5                       ; number of path elements
shstk   EQU     0DB00H                  ;shl, shell
shstks  EQU     4                       ; number of shell entries
shsize  EQU     32                      ; size of each shell entry
z3env   EQU     0DB80H                  ;env, z-system environment
z3envs  EQU     2                       ; size of env in records
z3msg   EQU     0DC80H                  ;msg, system message buffer
z3msgs  EQU     80                      ; size of msg in records
z3whl   EQU     0DCFFH                  ;whl, location of wheel byte
z3whls  EQU     1                       ; size of whl in bytes
```

Figure 4. The part of the file Z3BASE.LIB needed for the assembly of BYE.

C and the MS-DOS Screen Character Attributes

by Clem Pepper

When I bought a MS DOS computer as an addition to my CP/M system, I began digging through the manuals in a search for screen "escapes" similar to those I was accustomed to with my H-89. Surprise. This is one area where MS DOS appeared to have real shortcomings in my view. Through experience I have learned that there is more functionality available than might appear. This is available to us in a variety of ways. Making use of it requires more than a little homework, however.

Three possible approaches exist for controlling the video display with IBM PC family and compatibles. Each has its advantages and shortcomings.

The highest level approach is via the standard MS DOS service calls using interrupt 21H (INT 21H). With version 1 of MS DOS these were severely limited. With version 2.0 and higher an optional console driver, ANSI.SYS was added. The functions available are limited and relatively slow in execution, but provide the advantage of portability to any MS DOS machine.

The next level of access is by making calls directly to the ROM BIOS accessed via interrupt 10H. A wide assortment of functions are available, and execution is faster. Programs making calls to the BIOS will execute properly on all true compatibles but may cause problems with other MS DOS systems. My Zenith 161, described as 99 per cent compatible, operates correctly with all the INT 10H functions I am aware of.

The fastest level of access is by direct instruction to the hardware. While these provide the best performance there is also the greatest risk with portability.

The ANSI Functions

MS DOS provides cursor positioning and keyboard assignments using the ANSI escapes. The cursor functions are limited--the cursor cannot be turned off or its size changed, for example. Use of the ANSI functions requires a device driver for terminal emulation, ANSI.SYS. For this we must add the statement

Table 1: The ANSI screen and mode sequences.

Note: These control functions are available only with MS DOS 2.0 or greater. The statement DEVICE-ANSI.SYS must be included in your CONFIG.SYS file.

Definitions:

* n - a decimal number specified with ANSI characters.
* s - A decimal number used to select a subfunction.
* ESC - the ASCII code value 0x1BH.

Cursor Functions

CUP - Cursor Position ESC[n;nH - Moves the cursor to the row and column specified in the two parameters.

CUU - Cursor Up ESC[nA - Moves the cursor up n rows with no change in columns.

CUD - Cursor Down ESC[nB - Moves the cursor down n rows with no change in columns.

CUF - Cursor Forward ESC[nC - Moves the cursor forward n columns with no change in rows.

CUB - Cursor Backward ESC[nD - Moves the cursor backward n columns with no change in rows.

HVP - Horizontal and Vertical Position ESC[n;nf - Moves the cursor to the position specified by the parameters. Where none are given the cursor is moved to the HOME position. (Same as CUP.)

DSR - Device Status Report ESC[6n - The console driver outputs a CPR sequence on receipt of a DSR.

CPR - Cursor Position Report ESC[n;nR - Reports the current cursor position via the standard input in row and column sequence.

SCP - Save Cursor Position ESC[s - Saves the current position. It is restored with the RCP sequence.

RCP - Restore Cursor Position ESC[u - Restores the cursor position to the value it had on receiving the SCP request.

Erasing

ED - Erase Display ESC[2J - Erases the entire screen and homes the cursor.

EL - Erase Line ESC[K - Erases everything between the cursor and the end of the line.

Modes of Operation

SGR - ESC[s;...;sm - Invokes the graphic rendition specified by the parameter(s).

PARAMETER	PARAMETER FUNCTION	NOTES
0	All attributes OFF	normal white on black
1	Bold ON	high intensity
4	Underscore ON	monochrome displays only
5	Blink ON	
7	Reverse Video ON	
8	Cancelled ON	invisible
30	Black Foregound	
31	Red Foregound	
32	Green Foregound	
33	Yellow Foregound	
34	Blue Foregound	
35	Magneta Foregound	
36	Cyan Foregound	
37	White Foregound	
40	Black Background	
41	Red Background	
42	Green Background	
43	Yellow Background	
44	Blue Background	
45	Magneta Background	
46	Cyan Background	
47	White Background	

SM - Set Mode ESC[=sh or ESC[=h or ESC[=0h or ESC[?7h
SM invokes the screen width or type specified by the parameter.

PARAMETER	PARAMETER FUNCTION
0	40 x 25 monochrome
1	40 x 25 color
2	80 x 25 monochrome
3	80 x 25 color
4	320 x 200 color
5	320 x 200 monochorme
6	640 x 200 monochrome
7	wrap at end of line (wordwrap)

AH	OPERATION	OTHER INPUT REGISTERS	RETURN REGISTERS
0	Set Screen Mode	AL must be set equal to a value shown in Table 2b.	None
1	Set Cursor Type	starting scan line CH bits 0 - 4 CH bits 5 - 7 = 0 ending scan line CL bits 0 - 4 CL bits 5 - 7 = 0	None
Note: It is standard to use scan line numbers 0 - 7.			
2	Set Cursor Position	DH,DL = row, column (0,0 = home) BH = page number (0 for graphics mode)	None
3	Read Cursor Position	BH = page number (0 for graphics mode)	DH,DL = row, column CH,Cl = current cursor mode
5	Select Active Display Page (Text mode)	AL = new page value (0-7 for modes 0 and 1) (0-3 for modes 2 and 3) (0 for graphic modes)	None
6	Scroll Active Page Up	AL = number of lines to scroll selected window. AL = 0 erases the entire window. CH,CL = row, column of upper left-hand corner of the scroll. DH,DL = row, column of lower right-hand corner of the scroll. BH = attribute byte for the erased line.	None
7	Scroll Active Page Up	AL,CH,CL,DH,DL,BH have same functions of AH = 6	AL = character read
8	Read Character and Attribute at Current Cursor Position	BH = display page (for text modes)	AH = attribute of AL
9	Write character and Attribute at Current Cursor Position	BH = display page (for text modes) BL = attribute of char (Text) or color of char (graphics). CX = count of chars to write. AL = char to write.	None
10	Write character Only at the Cursor Position	BH = display page (for text modes) CX = count of chars to write. AL = character to write.	None
14	Write Character to Screen, Advance Cursor	AL = character to write. BL = foreground color BH = display page (Text)	

Table 2a The interrupt 10H functions. Only those functions relevant to screen activities are included.

MODE	TEXT/ GRAPHICS	RESOLUTION	UTILIZES
0	Text	40 x 25 display/monochrome	Color Card
1	Text	40 x 25 display/color	Color Card
2	Text	80 x 25 display/monochrome	Color Card
3	Text	80 x 25 display/color	Color Card
4	Graphics	320 x 200 pixel graphics/color	Color Card
5	Graphics	320 x 200 pixel graphics/monochrome	Color Card
6	Graphics	640 x 200 pixel graphics/monochrome	Color Card
7	Text	80 x 25 display/monochrome	Monochrome Card

Table 2b. Video modes available with the interrupt 10H functions.

```
DEVICE=ANSI.SYS
```

to our CONFIG.SYS file. Note, however, that we must be using MS DOS 2.0 or greater.

In addition to cursor functions we can use the ANSI functions for screen clearing, invoking graphics renditions, setting the screen mode, and reassigning the keyboard. Table 1 is a listing of the cursor, screen, and graphics functions. While the functions operate slowly and do not address all of our concerns they do have the advantage of portability.

Also, ANSI provides a means of bypassing DOS's inherent practice of writing white characters on a black background regardless of the screen color settings from the BIOS. More on that later.

Program ANSISCRN.C (Listing 1) employs escape sequences for clearing the screen and positioning the cursor. Don't be surprised if no response is obtained from some. My own computer responds to the cursor UP and DOWN escapes but ignores positioning along a line. Which, among other reasons, explains why I wrote the procedures found in the header file, DOS_UTIL.H (Listing 2).

```
/* ANSISCRN.C
** A program illustrating the ANSI screen functions.
*/

#include <stdio.h>

#define CLRSCRN       ''\033[2J''     /* clear screen, home cursor */
#define UP            ''\033[5A''     /* move cursor up five rows  */
#define DOWN          ''\033[6B''     /* move cursor down six rows */
#define RIGHT         ''\033[10C''    /* move cursor right 10 cols */
#define LEFT          ''\033[8D''     /* move cursor left 8 cols   */
#define SAVE          ''\033[s''      /* save the cursor position  */
#define RESTORE       ''\033[u''      /* cursor return to save pos */
#define POSITION      ''\033[10;5f'' /* cursor to row 10, col 5   */
#define CUR_POS_REQ   ''\033[6n''     /* request cursor position   */
#define CUR_POS_RPT   ''\033[n;nR''   /* report cursor position    */

main()
{
  puts(CLRSCRN);
 printf(''Press any key to continue until return to DOS.\n'');
 getch();
 printf(''Position cursor at row 10, col 20'');
  puts(POSITION);
   getch();
 printf(''Move the cursor up by five rows.'');
  puts(UP);
   getch();
  puts(SAVE);
 printf(''Now move the cursor down by six rows.'');
  puts(DOWN);
   getch();
 printf(''Move the cursor to the right 10 cols'');
  puts(RIGHT);
   getch();
 printf(''Move the cursor to the left 8 cols'');
  puts(LEFT);
   getch();

 printf(''Position cursor at row 20, col 1'');
   getch();
  puts(''\033[20;1f'');
  /* ** The report prints on the screen and exits the program ** */
  printf(''Request and display the cursor's present position.'');
   getch();
  puts(CUR_POS_REQ);
}
```

Listing 1.

Using INT 10H With The 8088/80x86 Registers

Although I do make occasional use of an ANSI function I have found the use of the BIOS interrupt functions to be considerably better. With these we can achieve most of what we need to do quickly and expeditiously. But first we will have to do our homework. In this article we will learn how to use the video I/O functions of interrupt 10H (INT 10H). INT 10H provides 16 video screen I/O functions based on a value assigned to CPU register AH. Table 2 defines 10 character related functions.

Access to the AX register is provided through library function int86. This function is in the libraries provided with Mix's Power C, Borland's Turbo C, and the Microsoft compilers. These are compilers I know of there are certain to be others. With this function our program must #include <dos.h> for the function declaration and definition.

Some knowledge of the 8088/80x86 registers is needed as registers other than AX are required for the functions. Figure 1 diagrams the four general purpose registers: AX, BX, CX, and DX. These are not the only registers, of course, there are others. But these four are appropriate to the video screen functions.

Each register is 16 bits wide, as shown. However each can also be addressed as an 8-bit register by use of H and L to designate the High or Low half respectively.

There are a few conventions with respect to register usage.

* The content of AH determines the call type. The INT10H video function numbers are placed in AH.
* The character or pixel value is placed in AL.
* The content of the BX, CX, and DX registers are preserved across the calls.
* For graphic functions the column number is passed in CX; the

Listing 2

```
/* DOS_UTIL.H
** Cursor and screen utility functions for use with
** video screen programs.
*/

#include <stdio.h>
#include <conio.h>
#include <dos.h>

/* uncomment only if needed with your compiler */
#define VIDEO 0x10  /* interrupt 10H - screen functions */
#define SFKEY 0x16  /* interrupt 16H - keyboard I/O     */

/* #define OUTPORT1 0x3B4  uncomment for MDA adapter */
/* #define OUTPORT2 0x3B5  uncomment for MDA adapter */
#define OUTPORT1 0x3D4    /* comment for MDA adapter */
#define OUTPORT2 0x3D5    /* comment for MDA adapter */

 /* === Utility function global declarations === */
int col;        /* column variable, general screen  */
int colul;      /* column variable, upper left cor  */
int collr;      /* column variable, lower right cor */
int row;        /* row variable, general screen     */
int start_row;  /* beginning row definition         */
int rowul;      /* row variable, upper left cor     */
int rowlr;      /* row variable, lower right cor    */
int lines;      /* lines to be cleared              */
int chr_attr;   /* assign attribute                 */
int asc;        /* ASCII code value                 */
int scn;        /* SCAN code value                  */
int cur_no;     /* sets number of cursor lines      */
int mode;       /* set to available mode            */
int page = 0;   /* typical value. can be changed in */
                /* program when required.           */
int set_bk = 0; /* value for background set         */
int set_pal;    /* ID of color palette, 1 or 2      */
int bk_gnd;     /* set background color             */
int fr_gnd;     /* set foreground color             */
int row_no;     /* row position number              */
int col_no;     /* column position number           */
int sav_col = 0;
int sav_row = 0;
int kbd_f = 0;  /* keyboard display status; 1 = on  */
int b_flg = 0;  /* when set, beep is silenced       */

char chr;       /* char to be displayed with attr   */

/* ==    turn off the cursor    == */
curoff()
{
      outport(OUTPORT1,0x0A);
      outport(OUTPORT2,0x20);
}

/* == turn on the cursor == */
curon()
{
      outport(OUTPORT1,0x0A);
      outport(OUTPORT2,0x06);
}

/* == set the cursor size == */
cur_size()
{
      outport(OUTPORT1,0x0A);
      outport(OUTPORT2,cur_no);
}

/* == position cursor at row and col values == */
void pos_cur(col,row)
{
  union REGS regs; /* dos.h union         */
    regs.h.ah = 2; /* set cursor position */
    regs.h.dh = row;
    regs.h.dl = col;
    regs.h.bh = page;   /* video page no. */
     int86(VIDEO, &regs, &regs);
}

/* == read cursor row and column values  == */
int rd_cur_pos()
{
 union REGS regs;   /* dos.h union             */
  regs.h.ah = 3;    /* get cursor position     */
  regs.h.bh = page; /* typically zero          */
   int86(VIDEO, &regs, &regs);
  /* transfer register values to memory */
    row_no = regs.h.dh;  /* row number  */
    col_no = regs.h.dl;  /* col number  */
}

/* ==     set the video mode     == */
void set_mode(mode)
{
  union REGS regs; /* dos.h union         */
    regs.h.ah = 0; /* set mode            */
    regs.h.al = mode;
     int86(VIDEO, &regs, &regs);
}

/* == set the attribute and display char == */
void set_atr(chr)
{
 union REGS regs;       /* dos.h union           */
  regs.h.ah = 9;        /* set attr function no. */
  regs.h.ch = 0;
  regs.h.cl = 1;        /* display char 1 time   */
  regs.h.al = chr;      /* char to be displayed  */
  regs.h.bh = 0;
  regs.h.bl = chr_attr; /* desired attribute     */
   int86(VIDEO, &regs, &regs);
}

/* == screen clear == */
void clr_scrn_all(int chr_attr)
{
 union REGS regs;       /* dos.h union           */
  regs.h.ah = 6;        /* scroll up 25 lines    */
  regs.h.al = 0;        /*  when al is = 0.      */
  regs.h.ch = 0;        /* top row               */
  regs.h.cl = 0;        /* upper left screen col */
  regs.h.dh = 24;       /* row to scroll up from */
  regs.h.dl = 79;       /* col to scroll from    */
  regs.h.bh = chr_attr; /* attribute byte        */
   int86(VIDEO, &regs, &regs);
}

/* == screen partial clear, scroll up == */
void clr_up(lines,rowul,rowlr,colul,collr)
{
 union REGS regs;       /* dos.h union       */
  regs.h.ah = 6;        /* scroll up         */
  regs.h.al = lines;    /* these lines       */
  regs.h.ch = rowul;    /* upper left row    */
  regs.h.cl = colul;    /* upper left col    */
  regs.h.dh = rowlr;    /* lower right row   */
  regs.h.dl = collr;    /* lower right col   */
  regs.h.bh = chr_attr; /* attribute byte    */
   int86(VIDEO, &regs, &regs);
}

/* == screen partial clear, scroll down == */
void clr_down(lines,rowul,rowlr,colul,collr)
{
 union REGS regs;       /* dos.h union       */
  regs.h.ah = 7;        /* scroll down       */
  regs.h.al = lines;    /* these lines       */
  regs.h.ch = rowul;    /* upper left row    */
  regs.h.cl = colul;    /* upper left col    */
  regs.h.dh = rowlr;    /* lower right row   */
  regs.h.dl = collr;    /* lower right col   */
  regs.h.bh = chr_attr; /* attribute byte    */
   int86(VIDEO, &regs, &regs);
}

/* == read function or other non-ASCII key == */
int rd_nonasky()
{
 union REGS regs;   /* dos.h union             */
  regs.h.ah = 0;    /* ASCII code will be here */
  regs.h.al = 0;    /* SCAN  code will be here */
   int86(SFKEY, &regs, &regs);
  /* transfer register values to memory */
    asc = regs.h.ah;  /* ASCII  code */
    scn = regs.h.al;  /* SCAN   code */
}
```

```
/* VID_ID.C
** A program to test for video adapter in use, then
** logically select the correct address for cursor
** ON/OFF and shape control.
*/

#include <stdio.h>
#include <dos.h>
#include ''dos_util.h''

#define UCGA  ''This system has a CGA adapter.''
#define UMDA  ''This system has a MDA adapter.''
#define UNKN  ''This system has an unrecognized adapter.''

  /* === ---- global declarations ---- === */
int row = 0;
int col = 0;

/* ==    MDA adapter cursor off    == */
mda_curoff()
{
   outport(0x3B4,0x0A);  /* port out to 6845 addr reg   */
   outport(0x3B5,0x20);  /* cursor off to 6845 data reg */
}

/* ==    CGA adapter cursor off    == */
cga_curoff()
{
   outport(0x3D4,0x0A); /* port out to 6845 addr reg   */
   outport(0x3D5,0x20); /* cursor off to 6845 data reg */
}

/* ==    MDA adapter cursor on     == */
mda_curon()
{
   outport(0x3B4,0x0A); /* port out to 6845 addr reg   */
   outport(0x3B5,0x06); /* cursor on to 6845 data reg  */
}

/* ==    CGA adapter cursor on     == */
cga_curon()
{
   outport(0x3D4,0x0A); /* port out to 6845 addr reg   */
   outport(0x3D5,0x06); /* cursor on to 6845 data reg  */
}

/* ** ==     Begin program      == ** */
main()
{
int cga_flg = 0, mda_flg = 0;
unsigned vdap = 0, vmod;

 /* ** clear screen and home cursor ** */
  clr_scrn_all(0x7);
  pos_cur(col,row);

 /* ** peek useage: int peek(int segment,unsigned
offset);** */
  vdap = (peekb(0x40,0x10));
  printf(''video memory value is %d.\n'',vdap);

 /* ** mask off all bits except 5 and 4 ** */
   vmod = 48&vdap;
   printf(''masked integer value is %d.\n'',vmod);

 /* ** inform user of video adapter in use ** */
 /* **  and select MDA or CGA cursor       ** */
  if(vmod == 32)  {
     puts(UCGA);
      cga_flg = 1;
                  }
  else if(vmod == 48)  {
     puts(UMDA);
      mda_flg = 1;
                       }
  else puts(UNKN);

  if(cga_flg == 1) cga_curoff();
 else
  if(mda_flg == 1) mda_curoff();

    printf(''Press any key to continue.\n'');
     getch();

  if(cga_flg == 1) cga_curon();
 else
  if(mda_flg == 1) mda_curon();

   exit(0);
}
```

Listing 3.

```
/* PRN_TEXT.C
** A program for setting the CGA background color,
** positioning the cursor, displaying text in
** a foreground color.
*/

#include ''dos_util.h''

/* ANSI function to bypass DOS's white on black screen */
#define SCREEN ''\033[37;44m''

extern int chr_attr;
extern int row;
extern int col;
extern char chr;

/* == Begin program == */
main()
{
int i = 0;
char chr[] = { ''Hi Y'all!\0" };

 /* ** clear screen and set background to blue with ** */
 /* **  white foreground                           ** */
  chr_attr = 0x1F;
  clr_scrn_all(chr_attr);
   puts(SCREEN);              /* DOS white on blue also */

 /* ** positionthe cursor ** */
  col = 10; row = 5;
  pos_cur(col,row);

  /* ** display text on screen with green background ** */
  /* ** and one blinking character                   ** */
   while(chr[i] != '\0')          {
   if(chr[i] == 'Y') chr_attr = 0xAF;
   else chr_attr = 0x2F;
   set_atr(chr[i]); i += 1;
    col += 1;  pos_cur(col,row);   }

   exit(0);
}
```

Listing 4.

```
/* MONO_ATT.C
** A program for setting mono attributes,
** positioning the cursor, and changing the
** cursor size.
*/

#include ''dos_util.h''

extern int chr_attr;
extern int row;
extern int col;
extern char chr;

main()
{
int i = 0;
char chr[] = { ''How's it goin', Pal?\0" };
 /* ** clear screen and position the cursor ** */
  chr_attr = 0x7;  /* white foreground */
  clr_scrn_all(chr_attr);
  col = 10; row = 5;
  pos_cur(col,row);

  /* ** display text on screen with reverse video,   ** */
  /* ** intensified chars, and a blinking character  ** */
   while(chr[i] != '\0')          {
   if(chr[i] == '?') chr_attr = 0x87;       /* blinking */
   else if(chr[i] == 'H'|| chr[i] == 'P')
   chr_attr = 0x0F;                         /* intensified */
   else if(chr[i] == 'i' && chr[i+1] == 't')
   chr_attr = 0x70;                         /* reversed video */
   else chr_attr = 0x7;                     /* white on black */
   set_atr(chr[i]); i += 1;
    col += 1;  pos_cur(col,row);   }

   exit(0);
}
```

Listing 5.

```
/*
** CUR_VAR.C
** A program for scanning through eight cursor variations.
*/

#include ''dos_util.h''

extern int cur_no;

main()
{
  int n, k_flg = 1;
   char ans, cin;
    clr_scrn_all(0x7);
     cur_scan();

 do {
printf(''Is there a specific cursor you would like
        to see again? <Y/N>\n'');
   cin = toupper(getch());
   if(cin != 'Y')  k_flg = 0;
   else {
     keyinp(); cur_size();
     lng_deelay();  lng_deelay();
        }
   } while(k_flg) ;

 /* ** restore normal two bar cursor ** */
    cur_no = 0x06;  cur_size();
     exit(0);
}
cur_scan()
{
  cur_no = 8;
 /* ** starting with a single line, expand cursor
   to full block ** */
   printf(''Watch the cursor grow .... \n'');
   while(cur_no--)   {
    printf(''Sending %d to the 6845 data reg.\n'',cur_no);
     cur_size();
        deelay();     }
}
deelay()
{ unsigned i = 60000;
   while(i--); return;
}
int keyinp()
{
 char chrr = 0;
     which();  chrr = (getch());
      printf(''Cursor size is %c\n'',chrr);
      switch(chrr) {
     case '0': { cur_no = 0; sho_cur(); break; }
     case '1': { cur_no = 1; sho_cur(); break; }
     case '2': { cur_no = 2; sho_cur(); break; }
     case '3': { cur_no = 3; sho_cur(); break; }
     case '4': { cur_no = 4; sho_cur(); break; }
     case '5': { cur_no = 5; sho_cur(); break; }
     case '6': { cur_no = 6; sho_cur(); break; }
     case '7': { cur_no = 7; sho_cur(); break; }
      default: break;
                    }
}

which()
{
  printf(''Enter a number 1 - 7 for desired cursor\n'');
    return;
}

lng_deelay()
{deelay(); deelay(); deelay(); deelay(); return;  }

sho_cur()
{
    printf(''Sending %d to the 6845 data reg.\n'',cur_no);
     cur_size();
      lng_deelay(); return;
}
```

Listing 6.

```
/* MUNCH.C
** A simple animation program illustrating cursor control.
** Updated from the TOOLWORKS C original.
*/

#include ''dos_util.h''

#define MUNCHT  '' /o \\ ''
#define MUNCHB1 '' \\o / ''
#define MUNCHB2 '' \\o \\''

extern int row = 9;      /* row variable    */
extern int col = 5;      /* col variable    */

/* ** begin program ** */
main()
{
  int flg = 0, i = 1000, j = 70;
     clr_scrn_all(0x7);   /* clear the screen */

 /* ** turn off the cursor ** */
  curoff();

   while(j--)   {
    if(flg == 0)
              {
  pos_cur(col,row);   puts(MUNCHT);
  pos_cur(col,row+1); puts(MUNCHB1);
     deelay(i); col +=1; flg = 1;
              }
 else
              {
  pos_cur(col,row);   puts(MUNCHT);
  pos_cur(col,row+1); puts(MUNCHB2);
     deelay(i);  col +=1; flg = 0;
              }
                 }
 /* ** turn the cursor back on ** */
   curon();
    exit(0);
}

deelay(n)
int n;
{
   while(n--);
}
```

Listing 7.

row number in DX.

* For character (text) functions the column number is passed in DL; the row number in DH.

Several programs for this article utilize these registers in some capacity. Rather than include the required functions in each program individually I have lumped them in the utility file of Listing 2. Each program beginning with Listing 3 makes a call to this utility, #include "dos_util.h". The reason for the quotes is that I keep this file on my work disk. If you add it to your compiler's include file replace the leading and trailing "'s with < and >.

Your compiler library no doubt contains functions duplicating many of those in DOS_UTIL.H. That is good, and you should use them in your programs. But a study of the utility will provide insights in how to utilize the CPU registers for your own program designs.

The 6845 Video Controller

I interrupt the discussion on INT 10H because there is a need to touch on the 6845 here. This because of the way video memory is structured in the MS DOS system.

My earliest efforts in turning the cursor off and on consisted of reverse engineering an assembly language program that wrote instructions to the 6845's data register. At first nothing happened. Then, in a search for further information I found a different address for the data register in my Zenith "MS DOS Programmer's Utility Pack." Changing the address in my program made it work. Why remained a mystery for some time.

The reason, as I eventually discovered, is found in the way IBM deals with color versus monochrome video. Monochrome video occupies 4K bytes of memory beginning at location B0000H.

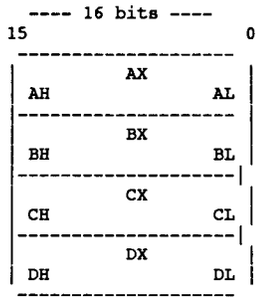


Figure 1. The 8088/80x86 general purpose registers.

```
    ** The attribute byte **
 ------------------------------
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
 ------------------------------
  7     - the blink bit
  6,5,4 - background color
  3     - high intensity
  2,1,0 - foreground color

    ** Color table listing **

COLOR     BACKGROUND       FOREGROUND
            6  5  4          2  1  0
Black       0  0  0          0  0  0
Blue        0  0  1          0  0  1
Green       0  1  0          0  1  0
Cyan        0  1  1          0  1  1
Red         1  0  0          1  0  0
Magneta     1  0  1          1  0  1
Brown       1  1  0          1  1  0
Lt Gray     1  1  1          1  1  1

 Note: Intensified brown is yellow.
       Intensified light gray is white.
       Intensified black is dark gray.
       Interchange foreground and background bits to
        obtain reverse video.
```

Table 3. The attribute byte with color definitions.

```
CODE  FOREGROUND     CODE  BACKGROUND
 30    black          40    black
 31    red            41    red
 32    green          42    green
 33    yellow         43    yellow
 34    blue           44    blue
 35    magneta        45    magneta
 36    cyan           46    cyan
 37    white          47    white

  ANSI useage - ESC[f;bm  where
    f = foreground code
    b = background code
    m = specifier (required)
```

Table 4. INT21H codes for setting screen character colors. (Source: Robert Jourdain, "Programmer's Problem Solver For the IBM PC, XT & AT," A Brady Book, Prentice-Hall Press, 1986, p154)

Memory for the CGA color adapter begins at B8000H; that for EGA at A0000H. The program I dissected was intended for a monochrome adapter (MDA). MDA and CGA 6845 register addresses are not the same. I have CGA and that is the address given in the Zenith manual.

Before running the example programs beginning with Listing 4 you must know which adapter you have. The cursor functions in DOS_UTIL.H using "outport" have 6845 register addresses for CGA. Program VID_ID.C (Listing 3) performs a test identifying the adapter in your system. If the program reports MDA you should change the addresses in DOS_UTIL.H as given for MDA in VID_ID.C. The EGA, which uses a video controller similar to the 6845 though not identical, uses either address depending on what card it is connected to. (If someone can enlighten me on a BIOS approach to cursor on/off control I will pass it on in a future article.)

Character Attributes

Inducing our program's characters to appear in color or to blink or intensify or flip into reverse requires a bit of doing. This because of the way MS DOS deals with video memory.

A screen having 25 rows of 80 columns each requires 2000 bytes of memory. Each byte contains the character for its associated screen position. For each character byte there exists an adjacent attribute byte. The content of this byte defines the properties of the displayed character. Table 3 describes the bit pattern of the attribute byte. (If you are a bit weak on binary to hexadecimal conversions here is the time to bone up.)

The default attribute is simply a white (actually light gray) character (foreground) on a black background. The default byte in hexadecimal is 0x7 (0000 0111). To intensify the character the byte is changed to 0xF. To display a blinking white (intensified light gray) character on a blue background the byte becomes 0x9F. These attributes for a CGA(or EGA) are demonstrated in program PRN_TEXT.C (Listing 4).

The line #define SCREEN "\033[37;44m" requires some explaining. The assignment chr_attr = 0x17; followed by the INT10H call to clear the screen (mandatory to convert the entire screen to blue) provides a blue background with a white foreground. Until MS DOS writes on it, that is. Then it is back to white on black. The statement puts(SCREEN); provides the same white foreground and blue background for the MS DOS characters. (They can be different colors, by the way. Whatever turns you on.)

PRN_TEXT should not be run with a monochrome adapter. (Keep in mind, however, that not all monochrome monitors are running with a MDA adapter. The built-in monitor in my Zenith is monochrome, but it responds to PRN_SET because the adapter is CGA. VID_ID will show you which adapter you have.)

The program MONO_ATT (Listing 5) is designed for the MDA adapter, provided you have modified DOS_UTIL's cursor routines. This program illustrates character blinking, reverse video and intensification. Note that the printing of the string is performed in a loop with selection logic for changing the attribute of a given character.

The next program, CUR_VAR (Listing 6), changes the cursor scan lines over the range of one to eight. It also provides for viewing any specific cursor size you select. This program will run with any adapter. A useful exercise would be re-writing this program to use INT10H function 1 (Set Cursor Type).

The last program, MUNCH (Listing 7), is a simple animation just for fun. Again, this will run with any adapter.

Summary

We have learned much of how our MS DOS computer goes about its business in writing to the video monitor. We have seen there are three levels of access to the video functions. The first, with the highest portability is by use of the ANSI.SYS functions. These require DEVICE=ANSI.SYS in your configuration file. If your version of MS DOS is 2.0 or higher ANSI.SYS should be on your distribution disk. The second level is by use of INT10H, which operates through routines in the ROM BIOS. For some MS DOS machines these may not be totally portable. The fastest, as well as the least portable, level is by sending instructions directly to the hardware. •

The following are excellent sources of related information.

Ray Duncan, "Advanced MS DOS", Microsoft Press, 1986

Robert Jourdain, "Programmer's Problem Solver for the IBM, XT & AT", A Brady Book, Published by Prentice-Hall, 1986

Paul Somerson, Editor, "PC Magazine DOS Power Tools - Techniques, Tricks and Utilities", A Bantam Book, June 1988

Nabajyoti Barkakati, "The Waite Group's Turbo C Bible", Howard W. Sams & Co., 1989

Forth Column

Lists and Object Oriented Forth

by Dave Weinstein

Last column I started talking about object oriented extensions to Forth. One of the drawing points of object oriented code is that the late binding scheme (used in the sample object oriented system) makes it possible to create object libraries which can be used and reused with little (if any) code modification. There is however, another way to get this kind of reusability, generic functions. As the name implies, generic functions are capable of acting on many different data types without modification. It is fairly simple to make Forth words generic (the wonders of a weakly typed language), the care must be taken when writing the generic routines so that all of the implicit information which the routine uses (i.e. the types are really pointers and are therefore one cell long) must be documented (these things are guaranteed to come back and bite you right before a do-or-die deadline if you aren't careful), and that any other information which the function needs is passed to it (i.e. the width of the cell and the size and number of dimensions for a generic array).

Probably the best way to show the differences is to write a reusable Forth module of some sort using first the generic model (many Forth words are in fact generic...they just don't call themselves that) since that is probably more familiar to most people, and then using the object oriented model. Since I've already stolen concepts from Modula-2, Pascal, C, and Smalltalk in the course of various columns, I'm going to continue this tradition and steal the heterogeneous list concept from LISP. Most of you are probably familiar with homogenous lists, the linked list in its varying forms (singly linked, doubly linked, circular, ESC) is one of the most common data structures used (I would put it second only to the array). But unlike a homogenous list, the items in a heterogeneous are not necessarily the same thing, they can be anything from numbers to strings to functions to other lists. Unlike LISP, however, we are going to only add two items together at a single time. If we are using generics, it makes sense to make the end-of-list pointer be the nil pointer (zero). In the case of generics, rather than having the special case logic embedded in a special sub-class (the empty-list being a subclass of the list), we put it in the words CONS, HEAD, and TAIL. A generic list also consists of the head and the tail, but the head is any one cell sized data structure

OLIST.SEQ

```
\               Heterogeneous List Package
\                 Object-Oriented Vers.
\
\ Messages:
\
\   <list-object> head : Return the lead item of the list
\   <list-object> tail : Return the tail of the list (itself
\                        a list)
\   <item-object> <list-object> cons : Return a new list
\   <list-object> empty? : Return whether or not the list is
\                          empty
\
\ Special Case:
\
\   nil-list : The empty list, this object is an empty list
\                    and marks the end of a list
\

\                       Import Object Code

from object-extensions  import message as message
                        import class as class
                        import end-class as end-class
                        import method: as method:
                        import end-method as end-method
immediate
                        import var as var
                        import subclass-of as subclass-of
                        import this-class as this-class
                        import self as self
end-imports

\                       Create Messages

message head   ( <list-object> -- <object> )
message tail   ( <list-object> -- <list-object> )
message cons   ( <object> <list-object> -- <list-object> )
message empty? ( <list-object> --  f | t )

class list
        var my-head
        var my-tail

        method: head
                my-head @
        end-method

        method: tail
                my-tail @
        end-method

        method: cons
                here  this-class ,  swap ( <item-object> ) ,
self ,
        end-method

        method: empty?
                false
        end-method

end-class list

class empty-list
        subclass-of list

        method: head
                self
        end-method

        method: tail
                self
        end-method

        method: empty?
                true
        end-method

end-class empty-list

empty-list nil-list
```

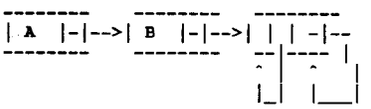
(which can hold a pointer to a larger structure if need be), and a one cell pointer to the next node in the list (or nil if it is the end). The object-oriented implementation, on the other hand, has the empty-list being a restricted sub-class of the list, and the logic is in the data structure (the object), not in the message which is sent. Functionally, the generic and object-oriented versions are the same (although in this case as with many small examples [the only kind which can easily be done in articles], there is inefficiency associated with the object-oriented version because of the overhead which the package requires).

Some brief notation

Before going into any more detail about lists, we need a common syntax to describe what we mean by lists. If, for example, we want to represent a list with two elements, A and B, we can represent them as:

```
( A B )
```

If we were to represent this in "box and pointer" notation, showing the pointers and the values, it would look like this:



The box-and-pointer notation more clearly expresses the relationships involved (for example, it shows the nil-list as the final element in the list), but it is too bulky for any more than the simplest explanations. If we want to explicitly show the nil-list at the end, we will adopt a slightly modified form of the first format, in which:

```
[ A B nil ]
```

is equivalent to:

```
( A B )
```

The Generic Version: Details

The generic version is fairly straightforward. Each node in the list is made up of two parts, the item and the pointer to the next node. This node will be nil if this is the last item in the list. The logic in the first two words (HEAD and TAIL) is equally straightforward; if the pointer to the list is nil, then that pointer is itself returned (the head of a nil-list is nil, and the tail of a nil-list is also nil), otherwise the head or tail of the node is returned. The word CONS in this example needs to create a block of memory, fill it with the information required to make it the new list. In a real situation, it would almost certainly use some sort of a memory management system, in this case we will use the standard Forth heap for ease of programming. So CONS saves the HERE pointer (this gives us the address of the new list, which CONS must return), and then "commas" the item and the pointer into the dictionary, creating the new list.

The Object Oriented Version: Details

This version is a bit more of a jump for most people than the previous example. Rather than putting the logic in the word HEAD or TAIL, we will put the logic inside of the data structure; the words HEAD, TAIL, and CONS only tell the data structure to act upon itself (they are messages, which tell the object to employ whatever methods it knows to perform the required action).

The creation of a new list is different in this case; rather than using a special case constant (e.g. nil), we use a special sub-class of list, the empty-list. As seen above, the head of an empty-list is also an empty-list, as is the tail, so the empty-list has its own methods for handling the methods TAIL and HEAD. But because an empty-list is simply a more specialized version of a list, it does not need to define the steps needed to create a list (it inherits this knowledge from its ancestral class).

GLIST.SEQ

```
\               Generic Heterogeneous List Package
\                  The Computer Journal: #42
\
\ Using the generics:
\
\   <list> head  :  Returns the <item> at the front of the
\                    list
\   <list> tail  :  Returns the pointer to the rest of the
\                    list (or nil)
\  <item> <list> cons  :  Returns the pointer to the newly
\                         created list
\   <list> empty?:  Returns whether or not the list is empty
\
\       End-of-list pointer:
\
\                  nil-list
\
\
\       Import the required Forth extensions

from records import record as record
             import element as element
             import end-record as end-record
end-imports

\       Define the structure of each node in the list

record list-node
        cell element item
        cell element tail^
end-record

\       Define the end-of-list pointer

0 constant nil-list

\       And now the handling code

: head  ( ^list -- item )
        dup nil-list <> if
                item @
        endif
        ;

: tail ( ^list -- item )
        dup nil-list <> if
                tail^ @
        endif
        ;

: cons ( item ^list -- ^new-list )
        here -rot swap ( item ) , ( ^list ) ,
        ;

: empty? ( ^list -- f | t )
        nil-list =
        ;
```

An example use of heterogeneous lists
or
What can we do with this anyway?

One of the most common programming constructs is the symbol, or look-up table. One of the uses is in binding a value to a function; we can do the same thing with a list of associated pairs (more jargon to confuse the unwary). Essentially, an associated pair is a list consisting of a value, and the object it is bound to. For our purposes, an associated list will be a character, and a function to which it is to be bound. So to search our "table" (in this case a very surreal table), we will need a list of associated pairs, a list of lists. An example of this notationally might look something like this:

```
( ( A Do-A ) ( B Do-B ) ( C Do-C ) ... )
```

Where A is a character and Do-A is the CFA of a Forth word.

We will keep a pointer to our control-character list in a variable, called CONTROL-CHAR^. Since originally there are no control characters in our list, we set the pointer to point to the empty-list:

EXAMPLE.SEQ

```
\              Example TypeWriter Program
\            Dave Weinstein - _The Computer Journal_
\

 variable control-char^  nil-list control-char^ !

: make-assoc    ( char ^func -- ^assoc-list )
    ( ^func ) nil-list cons
    ( ^char ) ( list ) cons
    ;

 : add-assoc     ( ^assoc-list -- )
    ( ^assoc-list ) control-char^ @ cons
    ( ^new-list ) control-char^ !
    ;

 : is-bound-to   ( <function> | char -- )
    ( char ) ' ( <function> ) make-assoc
    ( ^assoc-list ) add-assoc
    ;

 : is-done       (  ^list --  ^list f |  t )
    dup empty? if
       drop true
    else
       false
    endif
    ;

 : the-char      ( char1 ^list -- char ^list char1 char2 )
    over over  head head ;

 : the-func      ( ^list -- ^func )
    head tail head ;

 : do-character  ( ^func char ^list -- )
    begin
       is-done if
          swap ( ^func ) execute  ( Do default )
          true                    ( Exit loop )
       else
          the-char  = if
             the-func execute
             ( ^func ) drop  ( Drop default )
             true            ( Exit loop )
          else
             ( ^list ) tail  ( Next assoc-list )
             false           ( Continue loop )
          endif
       endif
    until
    ;

 : typewriter    ( -- )
    begin
       ['] emit          ( The default function )
       key             ( Get the character )
       control-char^ @ ( The control list )
       do-character    ( And do what needs to be done )
    again
    ;
```

```
variable control-char^  nil-list control-char^ !
```

Now we need to be able to turn a character and a function into an associated list, and then bind that on to our control list. First let's write a set of subsidiary words; MAKE-ASSOC which makes an associated list, and ADD-ASSOC which adds it to the control list.

```
: make-assoc    ( char ^func -- ^assoc-list )
        ( ^func ) nil-list cons
        ( ^char ) ( list ) cons
        ;

: add-assoc     ( ^assoc-list -- )
        ( ^assoc-list ) control-char^ @ cons
        ( ^new-list ) control-char^ !
        ;
```

With these words we can now define a word IS-BOUND-TO, which binds a character to a function:

```
: is-bound-to   ( <function> | char -- )
        ( char ) ' ( <function> ) make-assoc
        ( ^assoc-list ) add-assoc
        ;
```

Now what we need to do is to be able to search our list, to find a match if one exists; if it exists we execute the associated function, if it does not we execute a default function, which is passed in. Before we do this we will need a few supporting words. One IS-DONE, is used to determine if we have exhausted all of the possibilities in the list.

It uses a word EMPTY? which is defined in both the generic and the object-oriented packages... By importing the empty-list check from the implementation of the list, we create a utility which uses heterogeneous lists, but does not require any knowledge of the implementation.

```
: is-done       (  ^list --  ^list f |  t )
        dup empty? if
                drop true
        else
                false
        ;
```

The next words we need are used to pluck the character and the function from the control list. The associated list is the head of the control list, and the character is the head of the associated list. But since the tail of a list is always a list, the function is the head of the tail of the associated list (this is the part that often confuses beginning LISP programmers). So we can define THE-CHAR and THE-FUNC as follows:

```
: the-char      ( char1 ^list -- char1 ^list char1 char2 )
        over over head head ;

: the-func      ( ^list -- func )
        head tail head ;
```

With this information, we can now write our search-and-execute word, DO-CHARACTER. This word loops through the control structure until it finds a match, and if so, it executes it. If it gets to the end of the structure without finding a match, it executes the default function it was passed.

```
: do-character  ( ^func char ^list -- )
     begin
          is-done if
               swap ( ^func ) execute  ( Do default )
               true                    ( Exit loop )
          else
               the-char = if
                    the-func execute
                    ( ^func ) drop  ( Drop default )
                    true            ( Exit loop )
               else
                    ( ^list ) tail  ( Next assoc-list )
                    false           ( Continue loop )
               endif
          endif
     until
     ;
```

And now, with all of the underlying code written, we can write a simple typewriter. The control codes for this typewriter would be user defined and entered ahead of time by adding them to the control list. Since this is a typewriter, the default action will be to print (using EMIT) the character. So our typewriter looks like this:

```
: typewriter    ( -- )
     begin
          ' emit          ( The default function )
          key             ( Get the character )
          control-char^ @ ( The control list )
          do-character    ( And do what needs to be done )
     again
     ;
```

(Continued on page 35)

The Z-System Corner

by Jay Sage

A number of newer TCJ readers have commented that with this column they feel that they are coming into the middle of a very involved discussion that is hard to catch on to. Of course, one answer to that problem is for new TCJ readers to purchase back issues. I have been writing this column regularly since issue #25, and I am quite sure that all those back issues are still available. That solution notwithstanding, it is probably not a bad idea to stand back every so often and try to comprehend a larger picture. That is one of the tasks I will undertake this time.

Detailed technical content will not be forsaken entirely, however, since I regard that as the primary purpose of my column. At this point, I suspect that I am too much of a Z-System expert to talk about very many topics at a level that is appropriate for beginners. To serve their needs, I have been very actively soliciting articles from other authors. In this issue, for example, we have the first of the columns I promised a couple of issues back on how to set up a remote access system (aka bulletin board system) under the NZCOM autoinstall version of Z-System. Lee McEwen (aka Chris McEwen) has done a lovely job with that assignment.

The technical discussion this time will focus on some issues that arose in trying to install ZSDOS or ZDDOS on an SB180 computer with the XBIOS enhanced operating system. Before you say "But I don't have an SB180," let me assure you that the techniques have more general applicability. The specific XBIOS problem is one that has come up often and has been the source of considerable frustration to XBIOS users. [They are in good company, by the way. Just as I was finishing this article, I got a call from Bridger Mitchell about this very subject!] I am only sorry that it took me so long to get around to working on it. Gene Pizzetta, a fellow Bostonian, was the squeaky wheel that finally got my attention, and he has contributed a number of his own ideas to the solution.

Announcements

Before we get down to business, I have, as usual, a few announcements to make. First I would like to remind readers once again about Bill Tishey's superb collection of help files for the hundreds of Z-System programs now available. Bill can now generate diskettes in many formats besides Apple (using his son's Commodore 128), and he is willing to fill your diskettes with the files for only $10. My column in issue #36 gave the following procedure to follow: (1) send enough formatted diskettes (plainly labeled with the format) to hold at least 1000K bytes (up from 800K back then); (2) use a reusable disk mailer or enclose a mailer suitable for returning the diskettes to you; and (3) enclose a return address label, return postage, and the $10 copying fee. Bill's address is 8335 Dubbs Drive, Severn, MD 21144. If you prefer (or if you need 96-tpi, 8" SSSD, or NorthStar hard-sector formats), you can send the diskettes to me as well.

Second, I would like to make a special point of calling your attention to the GEnie RoundTable discussions that take place every Wednesday at 10pm Eastern time. The first such session of each month is devoted to Z-System, and I am the moderator, so this is your chance for a real-time dialogue with me. Go to page "685;2" on GEnie and enter "Room 2".

There are several changes to report in the roster of Z-Nodes. Regrettably, Bob Paddock's node #38 in Franklin, PA, has gone off the air. To offset that loss, however, node #73 in the St. Louis, MO, area has come back to life after being down for several years. Sysop George Allen and cosysop Walt Stumper would be happy to hear from you at 314-821-1078 (PC-Pursuit MOSLO/24). The equipment is currently a Xerox 820-II with a 10 Meg drive, but the sysops hope to expand soon to a 30+ Meg Ampro.

On the Z-Node front, I am also sorry to report that Z-Node Central (Lillipute) was downed by hardware failures on both computers! They have been off the air for a couple of months already as I write this, and sysop Richard Jacobson has just faced the truth: that it will not be coming back. Ladera Z-Node (#2) in Los Angeles will take over as Z-Node Central. Chicago area callers looking for Z support should check out the Antelope Freeway system run by ZDOS-coauthor Carson Wilson for CFOG (Chicago area FOG). This is one of a small number of remote access systems running under the Z3PLUS flavor of Z-System. The phone number is 312-764-5152 (PC-Pursuit ILCHI/24). We expect that its 'System One' will soon be a Z-Node ('System Two' supports MS-DOS).

Finally, there have been some very significant developments with BDS C. Leor Zolman completed some major additions to the Z version (BDS Z), and the final release has just gone out as I write this column in mid October. Programs generated by BDS Z now have a full Z-System header and can be linked as type-3 programs to load and run at an arbitrary address. ZDOS coauthor Cam Cotrill has already released a substantial amount of BDS Z code for performing the functions in the SYSLIB, VLIB, and Z3LIB assembly-language libraries that are not already built into BDS Z.

Leor has now turned over all of the marketing and some of the development responsibility for BDS C to me. Recognizing that the $90 price tag of the full package, however reasonable for what one gets, is an impediment to new users who want to experiment with C, we have prepared a low cost introductory package that (1) includes only one version of the code (either standard CP/M or Z-System), (2) contains only the essential files, and (3)

Jay Sage has been an avid ZCPR proponent since the very first version appeared. He is best known as the author of the latest versions 3.3 and 3.4 of the ZCPR3 command processor and for his ARUNZ alias processor and ZFILER file maintenance shell.

When Echelon announced its plan to set up a network of remote access computer systems to support ZCPR3, Jay volunteered immediately. He has been running Z-Node #3 for more than five years and can be reached there electronically at 617-965-7259 (on PC-Pursuit). He can also be reached by voice at 617-965-3552 (between 11pm and midnight is a good time to find him at home) or by mail at 1435 Centre St., Newton, MA 02159. Finally, Jay recently became the Z-System sysop for the GEnie CP/M Roundtable and can be contacted as JAY.SAGE via GEnie mail.

In real life, Jay is a physicist at MIT, where he tries to invent devices and circuits that use analog *computation to solve problems in signal, image, and information processing.*

comes with an abridged version of the manual (and without the fancy BD Software binder). This package will be offered for only $60. Other parts of the full package can be added later: $25 for the second version of the compiler, $25 for the support materials (RED editor, CDB debugger, and the parts of the manual covering them), or $40 for both at once. If the whole package is ordered at once, it comes complete with an attractive binder (also available with the introductory package for $5 extra).

It should be noted that BDS Z generates programs that run perfectly well under standard CP/M. Naturally, they will not recognize Z-System features like named directories, but they will accept the now standard DU: extended drive/user syntax instead of the older U/D: format of standard BDS C. The only disadvantage of using BDS Z rather than BDS C on a standard CP/M system is that the programs carry Z-System overhead (about 800 bytes) that don't provide them with any functionality.

What is a Microcomputer Operating System For?

The basic function of an operating system is to make one's life—one's computing life, that is—simpler. When microcomputers first came out, the biggest burden was dealing with the hardware. It was no fun for the computer user and programmer (largely synonymous in those days) to have to deal over and over with the intricacies of the physical operation of the hardware, such as getting characters to and from the terminal or paper tape reader/punch, not to mention the dauntingly more complex task of managing data on a magnetic tape or floppy diskette drive.

Gary Kildall's CP/M operating system provided a solution—and a very good one (by and large) in my opinion—to those problems. It did so by implementing a standardized and modular interface that handled the basic device communication tasks. CP/M, which stood (I believe) for "Control Program for Microcomputers," was the master program that one got running on the computer right after power up. It would then allow one to load and run other programs, with control always returning to the CP/M master program after each user program finished.

Besides accepting and interpreting commands issued by the computer operator, an operating system like CP/M also provides resident code (always ready in memory) for performing certain functions that application programs will often want to use. The simpler functions are things like sending a character to the terminal screen; the more complex ones include fetching from or writing to a floppy diskette the information associated with a logical entity known as a file.

With these functions implemented in the operating system code, application programs are easier to write and do not have to include the same code over and over. More importantly, they can run on a variety of hardware platforms, since the details of the physical hardware are handled by the operating system code, and the program can deal with things at a logical level.

Logical vs. Physical

Perhaps this is a good time for a brief aside on this matter of logical versus physical. We use the adjective "physical" when we are talking about things that are actually in the hardware. In the case of a floppy disk, for example, the physical items are the bits of data stored as magnetization patterns. These bits are grouped into sectors, and the sectors into tracks. In the case of a terminal screen, the physical items are the patterns of illuminated dots that we recognize as letters, numbers, and other symbols.

On the other hand, we use the adjective "logical" to describe those things which are essentially the creation of our minds (and programs). For example, there is no such physical thing as a "file." No matter how you examine a diskette, you will never find a file on it (as such); you will find only sectors and tracks. It is our choice to organize the data on the disk in a way that associates groups of such sectors with a file names and to store the file names in a particular group of sectors on the disk.

Modularity

CP/M is modular in the sense that it divides up the functions of the operating system into separate packages. One part is called the BIOS (basic input/output system). This part, which lives at the very top of the memory address space, deals directly with the hardware. It reads and writes physical sectors from and to a diskette; it determines whether or not a key has been pressed on the keyboard and, if so, which key; and it sends characters to the screen. The BIOS is the only part of CP/M that is different for each hardware implementation of a CP/M computer.

The second CP/M module is called the BDOS (basic disk operating system). It deals with logical constructs. We have already spoken of files. When a file is referred to, the BDOS figures out which physical tracks and sectors contain the data for that file. Another logical construct is lines of text. The BDOS has a function to send a complete line of text to the screen (as opposed to the BIOS, which can send only a single character), and it has another function to get a complete line of text from the user, allowing a limited amount of editing. These functions make it much easier for the application programmer to write his or her program.

The last CP/M module is called the CCP (console command processor). It gets a command typed by the user at the console and takes the appropriate action to carry out that command. Some commands, such as DIR or ERA, are implemented directly in the CCP code. Others require that a COM file be loaded from diskette and executed.

Command Processing Under CP/M

For the most part, CP/M accomplishes the functions it was designed to perform in admirable fashion. However, it was so concerned with solving the hardware interface problem (the programmer interface) that it devoted relatively little attention to the user interface. To be fair, it was born in the days when 16K of memory cost about $500 (in 1970s dollars, no less) and occupied an entire S-100 card (bigger by far than a whole SB180FX computer with 512K). Today we might not think that 64K is very much (some say that OS/2 feels dreadfully cramped in less than 3 Megs!), but it makes a lot of things possible that 48K (or even less) would not allow.

CP/M's command processor did little more than the minimum it was required to do, namely to run a few resident commands and to load external commands from disk. It did not provide many services to make the operator's life easier. You had to specify rather exactly the command you wanted performed; no leeway was allowed. And if you made a mistake, CP/M did not try to help; it just shrugged its shoulders and emitted a question mark.

The Niceties of Z-System

The Z-System has evolved over a period of nearly a decade now, but its goal from the very beginning has always been to make it easier and more convenient to operate the computer. My ideal is to have the computer do everything that it possibly can do for the user and leave to the user only those tasks that no computer could possibly figure out on its own. The command processor improvements I have introduced and the utilities I have written have all been directed toward that goal. I will now run through a short summary of Z-System features and try to indicate how they make the operator's life easier. This list is adapted from my book, *The ZCPR33 User's Guide."*

User Area Access

CP/M introduced the concept of disk "user" areas, which allowed the operating system to group files into separate logical directories (physically the files are all stored in the same directory, but they are tagged to indicate the user area). Unfortunately, CP/M provided no practical way to access files across user areas, which made them almost useless.

Back in the days when disks held only about 100K, there wasn't much need for this kind of organization, but today floppy

diskettes commonly have a capacity between 350K and 1.3 Meg. Hard disks with many tens of megabytes are also inexpensive and common. Under these circumstances, a single logical drive can hold hundreds or even thousands of files, and some way to organize them becomes essential.

Z-System makes it very easy and convenient to organize your files based on user numbers. Where CP/M allowed only a drive prefix to a file name (D:NAME.TYP), Z-System allows drive and/or user number prefixes (DU:NAME.TYP) so that files in other user areas as well as other drives can be referenced directly. In addition, Z-System allows meaningful names (similar to DOS subdirectory names) to be assigned to drive/user areas. This provides an interface that is far more suitable to the way people think and remember. With the DU: form, the operator has to think about the hardware (something he or she should not have to do, remember?); with named directories, the operator thinks in terms of function (TEXT: for text files, BDSC: for the C compiler, DBASE: for database files, and so on).

Terminal Independence and the Environment

While some would argue that the DOS hardware and software standards established by IBM's market dominance have resulted in an enforced mediocrity, there is no doubt that having a single environment in which to operate makes life much easier for applications programmers. Programs for DOS generally work right out of the box on any IBM compatible computer. Configuration is required only for fine-tuning.

CP/M, on the other hand, was designed to allow programs to run on an extremely wide variety of hardware. In those days, "personal" computer took on a different meaning—each person designed and built his own hardware. CP/M could be made to work with all of them, but elaborate configuration procedures were generally required, especially to match programs to the particular terminal used. To this day, we still have to deal with this hardware diversity.

What CP/M could have but failed to provide was a means for conveying to application programs information about the operating environment. Z-System has several modules that afford such communication. An area called the environment descriptor (ENV) contains information about the system configuration. Another system area called the message buffer (MSG) stores information that one program can leave for another program that runs later to read.

Part of the ENV is a section called the TCAP or Terminal-CAPability descriptor. The TCAP allows a program running under Z-System to determine the type of terminal in use and to adapt to the control codes it uses for special video operations. The ENV has information about the size of the screen and the printer's page. It also contains such information as the CPU clock speed and which disk drives are available (why allow attempts to log into drive C: if there is no drive C:—it often just hangs the computer). The Z-System supports many optional operating system features contained in optional modules, and the ENV contains information about these modules also.

The ENV and TCAP not only relieve the user of the nuisance of installing programs; they also make it very easy to change the installation. Suppose, for example, you want to print some files in 132-column mode instead of the usual 80-column mode. Under CP/M you might very likely have to get out a configuration program to redefine the printer setup. With a Z-System print utility, you would simply change the mode on your printer, run CPSET (console/printer set) to select the 132-column printer definition, and run the same print program as before.

Command Processing Enhancements

Under CP/M, you have to specify where the COM file to be run is located (otherwise the current drive is assumed). This is a perfect example of something that a computer can easily be smart enough to do for you, and Z-System does. As with modern versions of DOS (which took many years to catch on to this Z-System feature), you specify a list of directory areas that the operating system will scan for a requested COM file. If you wish (as you might when you know that your COM file is not on the search path), you can specify a directory using either the DU: prefix or the named directory DIR: prefix, and you are thus not limited to the current user area or the path.

With Z-System one is also no longer limited to issuing commands one at a time (DOS has been even slower to catch on to this). A single line of command input can contain a whole sequence of commands. As a result, you do not have to interrupt your thinking to wait for one command to finish before you can specify the second and subsequent steps in a process. You can work out a strategy for what you want to accomplish and issue all the commands at once, before you forget or get confused.

Many oft-repeated computational tasks involve sequences of commands (e.g., editing, assembling, linking, running; or editing, spell checking, printing). In such cases, the Z-System alias facility (similar in some ways to SUBMIT but far more flexible) can be used to define a new command name, which, when invoked, performs the entire sequence. This saves the user a lot of typing but more importantly eliminates the need to remember exactly what the sequence is. Basically, you solve the problem once and put the solution into an alias script. From then on, the computer is smart enough to take care of the complex details for you. I have given many examples of this in past columns.

Conditional Command Execution

There is only so much one can accomplish on a computer (or in life) without making decisions. Have you ever seen a programming language with no ability to perform tests and act in different ways depending on the results? Flow control (IF/ELSE/ENDIF) is unique to the Z-System command processor. Other operating systems that offer flow control at all limit it to operation inside a batch or script language.

A special set of Z-System commands can test a wide range of conditions, and the command processor will use the results of the tests to decide which subsequent commands will be performed and which will be skipped. This allows the Z-System to respond in a remarkably flexible and intelligent way. The solution to a complex computing task, one that requires on-the-spot decision-making, can be worked out once and embedded in an alias command. Then you won't have to tax your brain the next time you need to perform this task, and novice users will be able to do things on your computer that would have been beyond their own ability to figure out.

Command Processor Shells

If you do not want to deal with the operating system at the command level or if you want to have a command processor with different features, the Z-System shell facility allows you to install substitute user interfaces of your own choice at will. They can even be nested within each other.

Shells come in two common varieties: menu shells and history shells. The menu interfaces allow the user to pick tasks with single keystrokes and have the shell program generate the complex sequences of commands required to perform those tasks. The menu system shields the user from complexity, saves typing, and greatly reduces the chance of error.

History shells are enhanced command processors that remember your commands and allow you to recall and edit previous command lines. I wish the Apollo Domain minicomputer system I use at work (not to mention my DOS computer) had a history shell one quarter as nice as Z-System's LSH or EASE. They work like powerful wordprocessors on your command history, allowing searching and extensive editing.

What If You Make a Mistake

This is one of the other areas in which most operating systems behave in an

abominably primitive manner. When you issue a command that cannot be performed, they just issue an error message and then dump you back to square one. Often you are not even told what sort of error occurred (consider DOS's wonderfully helpful "bad command" message).

The Z-System behaves in a civilized manner under these circumstances. When an error occurs, the command processor turns the bad command line over to a user-specified error handler. The most sophisticated error handlers allow the operator to edit the command and thus recover easily from typing mistakes. In a multiple command sequence, if subsequent commands were allowed to run after an earlier command failed, there could be disastrous repercussions, and an error handler is indispensable.

The system environment even contains an error type, which the error handler can use to give you more specific information about what went wrong. It may be the familiar error of a COM file that could not be found, but there are many other possible causes for the difficulty. A file that you specified as an argument might not have been found (e.g., "TYPE FILENAM" when you meant "TYPE FILENAME"), or you may have specified an ambiguous file name to a program that cannot accept one (e.g., "TYPE *.DOC").

System Security

Like minicomputer and mainframe operating systems, the Z-System is a secure operating system. This means that it has mechanisms for limiting what any particular user can do or get access to. Dangerous commands (such as erasing, copying, or renaming files) can be disabled when ordinary users are operating the system but enabled when a privileged user is at work. Areas of your disk can be restricted from access for storage of confidential or other sensitive information. These security features come in very handy in the implementation of a remote access system or bulletin board (see Lee McEwen's article in this issue). There is no need for additional security to be provided by the remote interface program (BYE). The Z-System already includes a full suite of programs for regulating and controlling system security.

Summary

To sum it up, the goal of the Z-System is to provide an operating system that can be tailored extensively to user preferences and that can be made to handle on its own and automatically as many computational details as it can, leaving the user free to concentrate solely on those aspects of computer operation that require human intelligence.

Faking Out The System

For the technical part of this column, I would like to talk briefly about some techniques for adding extensions to a Z-System that it was not designed to accept. The need for this trick arose in connection with the installation of ZSDOS and ZDDOS (and their clock drivers) on an SB180 computer with the XBIOS enhanced BIOS, but it can be useful in other situations as well.

```
JetLDR for Z-Systems (ZCPR3), Version 1.00
Copyright (c) 1988 Bridger Mitchell

Syntax:
   JetLDR  [du:][library][.lbr]  member1.typ  member2.typ  ...
  or
   JetLDR  [du:]file1.typ  [du:]file2.typ  [du:]file3.typ ...

  ENV - environment                FCP - flow commands
  IOP - input/output               RCP - resident commands
  NDR - named directories          Z3T - terminal capabilities
  ZRL or REL - module in SLR or MS-relocatable (REL) format
      with member name: RCP, FCP, IOP, CCP, CP3, DOS, DO3, BIO, CFG or BSX

Notes:
  If first file is a library, extract remaining files from it.
  An ENV file must be the first loaded.
  Precede special modules (DOS, RSX, BSX, ...) with appropriate
CFG file.

Use Path: YES   Root Only: NO   Scan Current: YES   Explicit Directory: A0:

                        -------------------------
```

Figure 1. This is the internal help screen displayed by the command ''JETLDR //''. It shows how flexible a package loader JetLDR is.

XBIOS is a very nice and flexible system. One of its main features is that it keeps much of the BIOS in an alternate memory bank, leaving a much larger TPA (transient program area) for application programs than did the standard BIOS from MicroMint. The configuration and loading process, however, is somewhat unconventional (a forerunner in some ways to the NZCOM and Z3PLUS techniques).

The XBIOS system is loaded not from system tracks on the disk but from a file. This file is generated by a special utility program called SYSBLD (SYStem BuiLD) that allows one to define in a rather flexible way the configuration of one's personal Z-System, including the names of the CCP and DOS files to be used. Those component files, however, must be available in REL format, and the new Z-System DOS components are supplied in ZRL format only (because they have hooks to other parts of the system that can be resolved only by that format).

Changing Systems Using JetLDR

JetLDR is a lovely little utility written by Bridger Mitchell that knows how to load almost any module in a Z operating system. It is much faster and more careful than its predecessors, LDR and LLDR, and it is not limited to the non-code Z modules—such as the NDR (named directory register)—or to code modules preassembled for a fixed system—such as an RCP (resident command package) module FIXED.RCP. It can load code modules assembled in ZRL format to whatever address that module occupies in the current system and with all the hooks to other Z-System modules generated at load time. Thus MYRCP.ZRL, assembled once, can be used in any system configuration that allocates enough room for an RCP of that size.

Most remarkably, JetLDR can load even main operating system modules: CCP, DOS, or BIOS. Special adjunct configuration files (CFG) are used to help it in some of these specialized tasks (a little more about that later). JetLDR's internal help screen is reproduced in Figure 1 so you can see the whole list of modules it can handle. It is available from the usual Z suppliers for $20.

So, the obvious solution to the problem of getting ZSDOS or ZDDOS running under XBIOS is first to generate and boot a standard ZRDOS system (ZRDOS.REL comes with the SB180) and then to replace ZRDOS with, say, ZDDOS using the JetLDR command:

```
JETLDR ZDDOS.ZRL
```

ZSDOS can be loaded just as easily. On my system I have ARUNZ aliases that swap DOSs in a jiffy this way in case I want to perform some experiments.

There's The Rub

Now comes the problem. It's very nice that we now have ZDDOS or ZSDOS loaded and running, but if we want to take advantage of its wonderful time and date features, we must find a way to load its clock and (for ZSDOS) stamping module, too. The ZDOS utility SETUPZST makes it very easy to create the required loader, LDTIM.COM; the problem is: where can LDTIM put the driver code? [Aside: For those who own it, I am told that the DateStamper BSX module will work with ZSDOS, but I have not tried this myself. It requires no memory to load.]

In an NZCOM system, the MKZCM

system definition utility allows one to specify a "user buffer" area in memory, and this is just perfect for the clock/stamp module. ZDOS even has special facilities for taking advantage of this buffer. LDTIM can automatically determine the location of that buffer and install the drivers there, and a special patch to NZCOM (included with the ZDOS package) gives NZCOM the ability to reconnect the drivers automatically after a new DOS is loaded.

XBIOS's SYSBLD utility, unfortunately, does not support such a user buffer (this is true even in the 1.2 version that is able to load ZRL files). There is a way to trick the system into making some room for extra memory modules. This is to assign the extra memory space needed to one of the standard modules, such as the RCP. For example, if you use an RCP of the usual 2K (16 record) size and need one page (two records) of memory for a ZDDOS clock driver, you simply specify an 18-record RCP space. Then, when SETUPZST asks you for the address to which the clock driver should be loaded, you give it the starting address of the last page of this RCP space.

Once these steps have been followed, ZDDOS should be running with date stamping. ZSDOS could be installed similarly except that even more extra space would have to be allocated to the RCP. Although what I have described so far will get the system running, there is some danger that an oversize RCP could be loaded by accident and overwrite the clock driver. To prevent this, the ENV module should be patched to indicate that only the actual 16 records (10H) are available.

For those who do not face the problem of installing ZDOS on an XBIOS-equipped SB180, there are other uses of this kind of trick. For people who do not have the necessary tools (e.g., MOVCPM) to move the BIOS down to make room for special drivers (such as RAM disk drivers and special I/O boards), this same trick can be applied to open up protected-memory space for them. Other people may find it useful for quick experiments with special drivers before going to the trouble of moving the operating system around.

There is one final refinement I would like to mention. It is something I learned from Gene Pizzetta, who took my general recommendations above and worked out the details (see his file, ZD-XB11.LBR, available on many Z-Nodes). I have usually used either the IOP or RCP modules for this trick, but Gene recommended using the NDR instead. The reason for this is that the IOP, RCP, and FCP get allocated in 128-byte chunks, while the NDR gets allocated in much smaller 18-byte chunks, the space required for one name. If your clock driver takes, for example, 270 bytes (10EH), you would have to allocate three extra records, because the driver is a tiny bit over two records. If you steal space from an NDR, you can add just two records, but reduce the number of names in the NDR by 1.

Changing Command Processors

Generating a new CCP using JetLDR is a little trickier than changing the DOS. JetLDR could, as it does with a DOS or BIOS module, load the new CCP into its operating position in memory, but this would be of questionable value, since the CCP would survive only until the next warmboot. So, instead, when processing a CCP ZRL module, JetLDR normally writes the resulting absolute-code CCP to a file ZCCP.CCP (in the root directory, I believe).

This is where CFG files come into play. They are special code modules that JetLDR uses to perform special processing (see the file JLTOOLS.LBR on Z-Nodes for more detailed information). For example, CCPCFG.ZRL is one that tells JetLDR how to deposit the absolute CCP code that it generates directly into the XBIOS ram image of the CCP in banked memory (from which it is loaded on each warm boot). A similar CFG file could be written to tell JetLDR how to install the new CCP onto the system tracks of the current drive-A disk, but so far no one has done this. I would be happy to provide the CCPCFG module to XBIOS owners who would like it or to others who would like to use it as a model for writing other CFG files (send me a formatted disk with your copy of JetLDR, return mailer, etc.). •

SAGE MICROSYSTEMS EAST

Selling & Supporting the Best in 8-Bit Software

- New Automatic, Dynamic, Universal Z-Systems
  - Z3PLUS: Z-System for CP/M-Plus computers ($69.95)
  - NZ-COM: Z-System for CP/M-2.2 computers ($69.95)
  - ZCPR34 Source Code: if you need to customize ($49.95)
- Plu*Perfect Systems
  - Backgrounder II: switch between two or three running tasks under CP/M-2.2 ($75)
  - ZDOS: state-of-the-art DOS with date stamping and much more ($75, $60 for ZRDOS owners)
  - DosDisk: Use DOS-format disks in CP/M machines, supports subdirectories, maintains date stamps ($30 – $45 depending on version)
- BDS C — Special Z-System Version ($90)
- SLR Systems (The Ultimate Assembly Language Tools)
  - Assembler Mnemonics: Zilog (Z80ASM, Z80ASM+), Hitachi (SLR180, SLR180+), Intel (SLRMAC, SLRMAC+)
  - Linkers: SLRNK, SLRNK+
  - TPA-Based: $49.95; Virtual-Memory: $195.00
- NightOwl Software MEX-Plus ($60)

Same-day shipping of most products with modem download and support available. Order by phone, mail, or modem. Shipping and handling $4 per order (USA). Check, VISA, or MasterCard. Specify exact disk format.

Sage Microsystems East
1435 Centre St., Newton Centre, MA 02159-2469
Voice: 617-965-3552 (9:00am – 11:30pm)
Modem: 617-965-7259 (password = DDT)(MABOS on PC-Pursuit)

Embedded Controllers

A 68705 Application

by Joe Bartel, Hawthorne Technology

We don't always work with super micros at Hawthorne Technology. In fact some of our projects use processors at the opposite end of the scale. In January we built a very small controller based on a Motorola 68705. This is a good example of the kinds of small control projects that can be built with the new single chip micros.

A local company needed a device to control a random access video disk player for a show exhibit. The device would play a predetermined selection from the video disk for a fixed time when the observer pushed a button. The house lights had to dimmed while the video player was active. The actual control of the video player could be an extra cost RS-232 option or a port for a wired remote control port using a special pulse width modulated control sequence.

There were three main alternative designs for this project: 1) Use a PC, 2) Design it with PALs and TTL, or 3) Use a single chip micro. A normal PC could have been used and connected with an RS-232 interface, but this was rejected from consideration early because it was far too big and much too expensive. The hardware solution was rejected because of the effort needed for the design and the time required to debug it. Also this would have produced a very complex design with many chips. In many cases a special purpose chip can be purchased to do a particular function. When that is the case the hardware approach is very attractive. The other consideration was that there would have to be at least two different versions of the device built. This would double the effort of the hardware version. The last choice, a single chip micro was chosen.

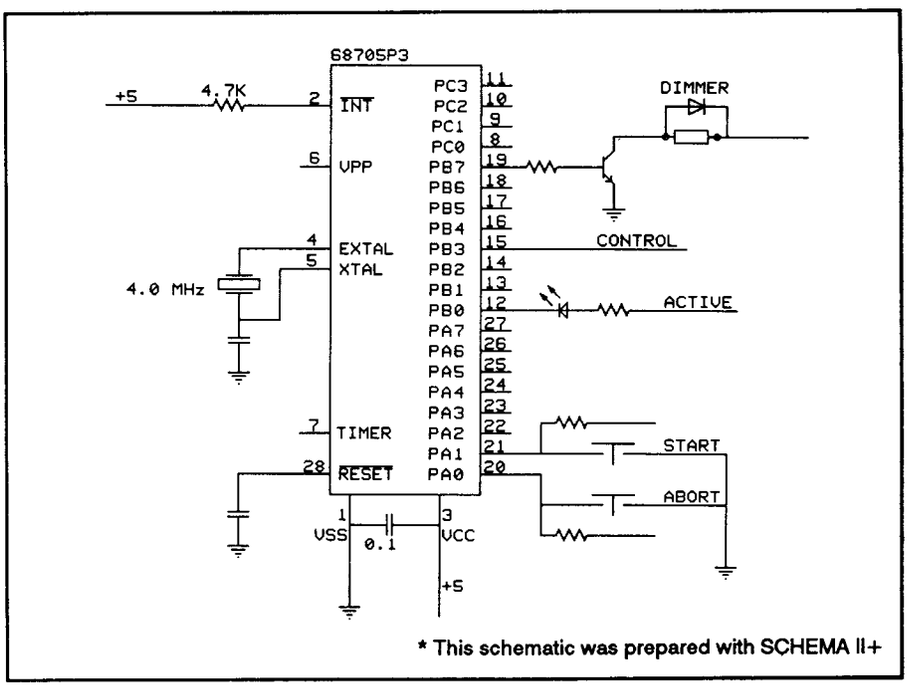
A single chip micro computer has all the needed parts to form a complete system on a single chip. There is a processor, some RAM, some EPROM, and a few peripheral devices. Different companies offer different families of single chip computers with a great deal of variety in what is offered. Some are larger and faster, others are smaller and slower. All of them offer a cost effective solution to a wide variety of problems. Almost every keyboard has one and many appliances like microwave ovens and televisions have one in control.

There were several other reasons why the 68705P3 was chosen. First, only a very simple program was needed. Second, the I/O that was present on the chip was adequate for the task. Third it involved a simple form of timing. The chips were available at a low cost and were very small. The chip used is only 28 pins like an EPROM, and they cost $15.00 at a local distributor. The final factor was that we had the capability to develop for the 6805 where we didn't have all the tools needed to use a different single chip family.

This was the company's first microprocessor based project. For a first project they wanted something that was very simple and safe so they could learn about the use of micros in general for controller applications. Three factors helped make this a very short and successful project. The task that had to be performed was simple and very well defined. There were no time constraints involved. All the information needed was available at the start.

The 68705 chip has several features which made it a good choice for this application.

- It has only 28 pins.
- An on board clock that only needs a crystal to operate.
- An on board power on reset circuit that only needs a timing capacitor to function.
- 20 pins that can be programmed as input or output. Of these pins 8 can drive LEDs or other higher current loads directly without a buffer.
- An 8 bit timer with a prescaler that can be used to provide a programmable



* This schematic was prepared with SCHEMA II+

periodic interrupt. To time more intervals software timers can be used.

- 112 bytes of RAM memory and 1804 bytes of EPROM memory.

The instruction set is very well suited for control applications, and programming for the 6805 family is very simple. In most cases it is not practical to do very much math with the chip. In comparison with the Intel 8748 or 8751 families the Motorola chip has a less complex set of instructions and will probably be easier for most programmers to work with.

Besides the usual load, store, add, and call instructions, there are four bit instructions that make control programming very easy. These are: BSET, BCLR, BRSET, and BRCLR. The first two set or clear a bit in a byte. The last two test a bit in a byte and branch if it is cleared or set. These are very useful in testing bits for switch inputs and turning bits on and off.

All the timing was done with software timers operating from a master interrupt tic as explained in an earlier article on software timers (see TCJ #30). The onboard timer and prescaler were set for a time tic of 512 usec.

The process for creating a programmed device follows several steps. First the program is written on an IBM PC using a text editor to create a source code file. Next the program is assembled with a cross assembler. Then the program is burned into a common 2732 EPROM. The contents of the EPROM is then transferred into the 68705 with a special programmer. The schematics for the 68705 programmer are published in many of the books on the 68705. If the program doesn't work (like most programs), then the micro can be erased like an EPROM and reused.

One feature that was added to the design was an LED that blinked once each second. This helped to trouble shoot the project in two ways. First if the LED was blinking we knew that the processor was running, the program had not crashed, and memory was not corrupted. Second we could tell by the blink rate whether or not our calculations for timing were OK. Many times a small indicator that the project is alive will pay off in much reduced trouble shooting time.

For many small projects a single chip micro like the 68705 can be a small and cost effective solution. They are small and cheap. Compared to all hardware designs they use fewer components. Also they can be reused if the device is no longer needed. Because they are so simple many projects can be designed in an afternoon. The simplicity also means that they can be hand wired and expected to work. If requirements, change then the program can be quickly changed. •

Editor

(Continued from page 3)

circuit board layout capabilities (and I probably will never need them).

I feel very comfortable with SCHEMA, although I am not yet completely familiar with all of its capabilities. It includes extensive libraries, plus the user can create modified or original libraries. Placing components, wires, and labels is quick and convenient. It produces a bill of materials and a wire list (great for wirewrapping), and does a design rule check. SPICE output is provided for circuit simulation, as are net/pin conversion programs for interfacing to popular CAE board layout programs.

I recommend SCHEMA if you need to prepare schematics. Contact them for their demo disk and literature.

CP/M Status

There is not much CP/M hardware available, and Chris McEwen's ad on page 38 may be about the last chance to pick up a bargain in distress priced new equipment. It's well worth the price even if you never use the 8086 portion. And NO, it is not IBM graphics compatible and will not run Lotus 1-2-3. But it does include a Z80 and a 10 Meg hard drive.

At one time AMPRO was very active, but I have not heard much about them lately. Does anyone have any feedback on what they doing? I have a number of their systems, in fact more than I can use. I would be interested in selling or trading an AMPRO Z80 Little Board in their bookshelf case with power supply, 5.25" floppy and 10 Meg hard drives, SCSI, up and running ZCPR3. I need a good two channel triggered scope. This is a good system for running Z-System and/or a BBS--I just have too many of them.

Laser Toner Cartridges

The daisy wheel and dot matrix printers see very little use since I got the Hewlett Packard LaserJet II. The quality and speed of the laser make everything else seem obsolete except for continuous feed labels. The only problem is that the per copy cost of the toner cartridge makes the output rather expensive. The cartridge is rated at 4,000 copies of a normal letter. Buying cartridges at the single piece street price of about $90 (including shipping) results in an estimated toner cost of $0.0225. The first cartridge gave out after only 2,800 copies of TCJ copy with bold titles and closely set type. This would be $0.0032 per page for toner. The cost of the laser and the paper are in addition to this. This is too high.

Various sources advertise a cartridge rebuilding service for under $50 which helps reduce the cost, but I wanted to find out what is actually involved. I contacted Chenesko Products (62 No. Coleman Road, Centerreach, NY 11720 (800)221-3516) for their literature. They offer a first-time recharge kit for $26.30 with refills for $19.75 (the price drops to $13.40 in quantity). At $19.75 for 2800 copies the cost is a much more attractive $0.0070.

I had heard horror stories of problems with recharged cartridges, but for a cost of less that 1 cent per copy I was willing to take a close look at it. I ordered a kit from Chenesko, recharged the cartridge, and am well satisfied with the results. The drum and other parts in the cartridge also wear out, so the cartridges cannot be recharged indefinitely.

Some caution is required because the printer can be permanently damaged if a poorly sealed cartridge dumps toner powder inside of the printer. Is it worth it? It all depends on the application. I feel that it is right for me, but what are your experiences? A more extensive article is planned, and your feedback is needed. •

Advanced CP/M

PluPerfect Writer and BDS C with REL Files

by Bridger Mitchell

PluPerfect Writer Updated

Full-screen text editors are perhaps the most widely used type of software on personal computers. In the CP/M world the step up from primitive line editors (remember ED?) began with WordMaster. That editor evolved into successive versions of WordStar and set a standard for on-screen formatting and printer control of edited text, using an 8-bit file format.

Another line of development flowed from the university computer science departments, where programmers were improving editing tools, initially for their own use. At MIT Richard Stallman conceived the EMACS editor. (At Harvard, Michael Aronson developed the MATE editor, which became the commercial product PMATE and is Jay Sage's favorite editor under both Z-System and DOS.) EMACS emphasized ASCII (plain-text) editing with the ability to work conveniently on multiple files and to automate complex changes in text. Formatting was left to other, stand-alone tools such as SCRIBE that provided extensive control over document organization and appearance.

Granddaddy EMACS has several descendants in the personal computer world. Only a few of them retain its highest-powered capability of dynamically modifying the editor itself—macros that can reprogram the program! But most provide at least these key features:

- virtual memory—edit files larger than physical memory
- multiple buffers—work on several files at one time
- multiple windows—simultaneously view different files, or sections of the same file
- short commands—designed for fast touch-typists
- text-oriented commands—allowing movement and deletions by character, word, sentence, paragraph
- file aids—directory, disk change, space on disk
- text-manipulation commands—rapid insert/delete/copy and search

Plu*Perfect Systems got its start—and its name—shortly after Kaypro began bundling Perfect Writer (PW) with its portable CP/M computers. Perfect Software, the publisher of PW, had developed the editor from MINCE—the first EMACS-like CP/M editor, which was published by Mark of the Unicorn, located just around the corner from MIT. Unfortunately, PW had been released with a few serious bugs. So, at the prodding of a number of users we found ways to correct them and then to add several new commands. Plu*Perfect Systems first published PluPerfect Writer (PPW) for the Kaypro version in 1983.

The Perfect Software company went on to develop an MS-DOS version of PW. Shortly thereafter it was acquired by Thorn EMI and discontinued all CP/M support. Also, Kaypro has now stopped supporting its former CP/M products.

PluPerfect Writer, however, has enjoyed a longer life, and continues to serve many users. In response to a number of requests we have finally updated PPW and are again able to supply it, this time in a generic version for all CP/M Z80 computers. It is best thought of as an upgrade to the original Perfect Writer version 1.20—PPW comes with limited documentation files on disk only, and users will benefit from consulting the Perfect Software user guide.

"We are all like ducklings," said a *Dr. Dobbs' Journal* article, "imprinted forever with our first experience in using a full-screen editor." For many, learning a different editor is unrewarding, even aggravating! Perhaps this is because a really good editor becomes so completely an extension of the writer that one is unaware of the individual keystrokes and commands. I imagine I'm no different in this regard; I just happen to have started CP/M life with MINCE, and grumble a lot if confronted with an editor that requires me to use function keys and work on just one view of a single one file.

Some of you may nevertheless be interested in trying PPW. Its keys can be "rebound" to more closely follow the editor you are accustomed to. It provides exceptionally useful two-window, multiple-file capability for working on programs. And with BackGrounder ii or other RSX's that provide keystroke macros it allows complex editing tasks to be programmed on the fly.

The Low-Down on HLL Support

I expect Jay Sage's column to cover news of an upgraded, Z-System-compatible version of the excellent C language compiler from BDS Software. When I first learned that Jay and author Leor Zolman were at work on this project I began thinking about the ways in which a high-level language (HLL) can relate to the Z-System.

Clearly, the language should make it convenient for the programmer to access the Z-System external environment. He should be able to write terminal-independent screen displays. File access should incorporate both the DU: and named-directory features, and should provide password control when used on remote or other secure systems. Leor and Jay have added all of this in the new 2.0Z version, making it the first HLL to directly support the Z-System.

It's also desirable to have the HLL be able to re-use the code in the existing SYSLIB, Z3LIB, VLIB and DSLIB libraries—routines that have been optimized and tested in many Z-System applications. And for a good number of applications, there are key sections of code that require assembler-level coding for speed. If the HLL provides a general-purpose interface to as-

*Bridger Mitchell is a co-founder of Plu*Perfect Systems. He's the author of the widely used DateStamper (an automatic, portable file time stamping system for CP/M 2.2); Backgrounder (for Kaypros); BackGrounder ii, a windowing task-switching system for Z80 CP/M 2.2 systems; JetFind, a high-speed string-search utility; DosDisk, an MS-DOS disk emulator that lets CP/M systems use pc disks without file copying; and most recently Z3PLUS, the ZCPR version 3.4 system for CP/M Plus computers.*

*Bridger can be reached at Plu*Perfect Systems, 410 23rd St., Santa Monica CA 90402, or at (213)-393-6105 (evenings).*

sembly-language routines, the user can have the best of each language—the HLL for rapid code development and management of complex data structures, and assembler for finely honed, compact and speedy subroutines.

Compiler Design Choices

But the marriage of HLL and assembler is not so quickly consummated. In designing a compiler, the programmer faces some very basic choices:

He can design the compiler to compile the source code into assembler statements, use a standard assembler to generate relocatable code, and then run a standard linker to produce the object code. This is the strategy Ron Cain followed with his famous Small-C, which Walt Bilofsky reworked into the solid C/80 compiler.

Or, he can compile directly into a standard relocatable format, and then use a standard linker. This is what Digital Research used in its PL/I compiler.

Or, he can compile into a custom relocatable format, and develop a custom linker to generate object code. This was Leor Zolman's choice for BDS C.

Think back to the very early days of CP/M. With your Digital Research package you got an incomprehensible manual, a few utilities like STAT and DUMP, and ASM—a small, fast, one-pass 8080 assembler that generated output in Intel hex format. Working in this primitive environment, Leor wrote the first versions of BDS C and polished them into a sleek, high-speed HLL compiler and linker.

At that time relocating assemblers and linkers for CP/M were just arriving. They were slow, quirky, and very expensive. So Leor took up the challenge and wrote his own linker and a pre-processor for the existing ASM assembler.

This design choice got an excellent, low-cost C compiler into users hands and promoted a wide range of major applications programming (including the MINCE and PW editors). But it stuck users with a kludgy method of mixing ASM and C routines. The ASM functions are coded in the normal way, but with macros to generate the function directory and entry addresses, as well as the external names used by each function, in the CRL format. There is an alternative—the ACRL assembler, written in BDS C and available from the C Users' Group. It generates CRL output from 8080 assembler statements with no macros.

"Why Mix C and ASM?"

High level languages have many, varied uses. They are excellent for quick coding of one-of-a-kind tasks and for prototyping algorithms, user interfaces, and major applications. I have frequently used them to develop configuration utilities that need to accompany the main application. But many times, in the CP/M world, the 64K memory limit and CPU speeds demand that key portions of a program be tightly coded in assembler.

I have used BDS C and the ACRL tools to write DATSWEEP and several other DateStamper applications, including the configuration utilities for PluPerfect Writer. Yes, you can beat your ASM code into the shape that is required to produce the right CRL format, but it's always messy. By contrast, by using the CP/M PL/I compiler my colleague Derek McKay produced applications such as MULTICOPY that nicely balanced HLL program organization and control-structure with exquisitely-timed disk formatting loops.

Each time I used BDS C with cobbled-together assembler files I thought, Wouldn't it be nice if BDS C could be used with regular REL files, much as PL/I can? Finally, with the arrival of version 2.0Z, I decided to investigate what would be required. As a first effort, it seemed that I should be able to write a format translation tool that would take a CRL file and turn it into a REL file. This would add one intermediate step to the BDS C production process, but it would make existing REL files usable and allow new assembler code to be written and assembled with standard assemblers. The REL files would then be linked together into a final COM file.

The BDS Dialect of C

BDS C is not a "full K&R" implementation of the C language. (Kernighan and Ritchie literally wrote "the book"—*The C Language*—which until the recent adoption of an ANSI standard was the definitive description of the language). It omits floating point and multiple-precision integer arithmetic (but library routines included with BDS C provide this capability, albeit in a non-standard way). Thus, it has just three data types: char (an 8-bit byte), int (a signed 16-bit word) and unsigned (16-bit word).

In K&R C external variables are declared "extern" and then referred to by name in the files that use them. They correspond most closely to public labels for variables in an ASM data segment. Probably because it is more complicated to link such files, the BDS C designer took an important shortcut in allocating memory for external variables.

There is no "extern" keyword as such; all externals are stored in a common data pool, either immediately following the code, or at a fixed address specified by the programmer. This means that if externals are used in more than one source file, they must appear in every file, in identical order. The recommended way to ensure this is to put all externals in a ".h" header file, and "#include" it at the top of every source file. This organization of variables most closely corresponds to a "blank common" data segment—a separate area of memory that is actually referenced by offsets from its base address, rather than by variable name.

A complete BDS C program consists of three components—(1) a runtime library that performs initialization and contains widely-needed low-level routines such as 16-bit multiply and divide, (2) the code segment containing the user's functions plus any standard library functions that it references, and (3) the external data segment (Fig. 1).

BDS C function names are upper-case 8-character. Thus it is essential that the relocatable code format support 8-character names. The good news is that the SLR Systems format does support 8-character externals, and, because it is a "byte-aligned" format, it's also relatively easy to debug and test.

The bad news is that the older Microsoft REL format appears to be limited to 7 bytes (for externals). So, the CRL2REL technique *requires* use of the SLR linker. However, I understand from discussing this with Al Hawley that, because Microsoft uses a three-bit field to define the length of a symbol, it is possible to instruct savvy assemblers and linkers to consider a "0-byte symbol" to be an 8-byte symbol. With further investigation it may be possible to extend CRL2REL to the Microsoft format also.

As some consolation, the SLR linker understands two dialects, and can accept Microsoft REL input files also. You can use files assembled with an assembler that produces MS REL output and mix them in the linkage step with SLR format. The MS REL routines will be limited to accessing other routines that have at most 7 character names.

The CRL2REL Translator

With this background and a detailed analysis of the formats of CRL and REL files, I was able to create CRL2REL, a short translator (written in BDS C, of course) that reads one or more CRL files and translates each into the corresponding REL file. For once, I'll omit the inner workings of this tool and turn directly to how to use it.

CRL2REL is run once, on the default BDS C libraries (DEFF.CRL, DEFF2.CRL)

```
CRL2REL deff deff2
```

to produce default DEFF.REL and DEFF2.REL files.

Also, the BDS C runtime library is (re)assembled once, to pro-

Figure 1. BDS C Memory Model

```
runtime library
      initialization
      memory parameters
      file control blocks ...
code segment
      main function
      other functions
''blank common'' segment
      external data area
free memory
```

Figure 2. Integrated Memory Model

```
runtime library
      initialization
      memory parameters
      file control blocks ...
code segment
      main function
      other functions
      assembler routines
data segment
      assembler routines' data
''blank common'' segment
      C routines' external data area
free memory
```

duce a REL output files (rather than the absolute C.CCC image file). For my own use I set the SLR assembler option to generate global labels, so that my ASM routines could refer by name to data and routines in the runtime library.

```
SLRMAC ccc.z80/r
```

These preliminary, once-only steps provide us with re-usable REL files of the runtime library (CCC.REL) and the default library routines (DEFF.REL, DEFF2.REL).

Next, compile each C source file in your application (with the CC compiler) into a CRL file,

```
CC myprog -e EEEE
CC mysubs -e EEEE
```

where EEEE is a suitable hex address for the external C data, and then translate it with CRL2REL into REL

```
CRL2REL myprog mysubs
```

Now link the REL files together to produce one object file

```
SLRNK+ myprog/n/p:100/m:4,ccc,myprog,mysubs,deff/s,
     deff2/s,.../e
```

Editor's note: The above line was folded to fit in the column width.

In this command line we name the output file MYPROG.COM (using the "/n" flag), with a starting address of 100h, request a symbol table file ("/m:4"), and take care to load the runtime library (CCC) as the first REL file, followed by the program and subroutine files. We also instruct the linker to search ("/s") the two default libraries. If you have other libraries to be searched include them here (e.g. "Z3LIB/s"). The final parameter ("/e") instructs the linker to exit by writing out the COM file.

One last step remains to make MYPROG.COM executable. The runtime library, linked from CCC.REL, needs the addresses of the external data area, the end of the program, and initialization for its stack pointer. This initialization is ordinarily done by the BDS C linkers (CLINK or L2). I wrote FIXCCC to perform the same functions:

```
FIXCCC myprog myprog
```

will read in its first argument as a REL file, and extract the address of the external data area from it. Then it reads the second argument as a COM file, installs the appropriate values, and writes out the modified file.

One behind-the-scenes "trick" that is used here is to have CRL2REL encode the EEEE value in the REL file, so that it can be passed on to FIXCCC and ultimately inserted into the complete object file.

To summarize, the standard BDS C procedure is:

```
CC progname -e EEEE
CLINK progname
```

The new procedure is:

```
CC progname -e EEEE
CRL2REL progname
SLRNK+ ....
FIXCCC progname progname
```

Postscript

When I embarked on the translator project I had in mind making it the testbed for a project that would directly upgrade the BDS C compiler itself. In principle, the compiler could directly produce REL output. The resulting file would have a code segment and a blank common segment for externals, and external references to each C or ASM function.

Further investigation, however, suggests this can't be accomplished easily, primarily because the BDS C code generator converts the external variables to absolute addresses at an early stage in the process.

The principal impact of this limitation is that BDS C cannot generate a relocatable, self-contained object file—a program that can be relocated to run at any reasonable address—because its external data area has been fixed at compile time. There are at least three good applications for a run-time relocation capability—a Type-4 Z-System program that the command processor relocates to run in highest user memory, a symbolic debugger such as the Kirkland debugger supplied with BDS C, and other resident system extensions.

However, Jay Sage has just recompiled and reassembled a version of all of the BDS C components to support Type-3 Z-System programs at a fixed runtime address of 0x8000. This should be a satisfactory substitute for Type-4s in many applications.

Turbo Pascal and ASM?

I'd hoped to close this column with thoughts about how assembly-language routines could be uses effectively with other high-level languages, and the obvious candidate is Borland's popular Turbo Pascal. Unfortunately, unlike the several C compilers available for CP/M, Turbo Pascal has no provision for linking together Pascal functions and procedures with separately assembled ASM routines. At least I have not been able to discover one, although I am not a regular Pascal programmer. Perhaps a reader more familiar with that compiler can invent a method of using ASM routines with Turbo Pascal. If so, do write!

Turbo Pascal does have a primitive capability to include machine-code bytes in the source file and have them executed as instructions. This may help in a few cases. If you can write a position-independent routine, or somehow determine where your bytes will be located in the object file, you can use a standard assembler to generate a machine-code listing and then type those codes into your Pascal source file. But in-line machine bytes are hardly a general ASM interface! •

Real Computing

The National Semiconductor NS32032

by Richard Rodman

By the time you read this, the Datacrime, AKA Columbus Day virus, will either have caused the destruction of Western civilization, or gone the way of Comet Kohoutek. Either way, it proves the sad truth that people who should know better, just plain don't.

The NS32 Trap Mechanism

Interrupts and traps which come into an NS32 are vectored through an interrupt table in almost the same manner as the CXP instruction. The PSR (flags), the MOD register (pointer to current module table entry), and the PC (program counter) are pushed on the stack. Then, the appropriate entry in the Interrupt Table is invoked. Each entry in the Interrupt Table is a *descriptor*, that is, a 16-bit module table entry pointer and a 16-bit code offset from the code base of the module indicated.

Every module in an NS32 system should have a module table entry somewhere in the first 64K bytes of stack. Each module table entry is 4 doublewords long: the first doubleword contains the static (data) base address, the second the link table address, and the third contains the code base.

The interrupt table can be located anywhere in memory. In a memory-mapped system, it must be located in the supervisor space. The first 10 entries are for specific purposes, in the following order: Non-vectored interrupt, non-maskable interrupt, abort trap (page fault from MMU), floating-point unit trap (a floating-point instruction executed with no FPU, or some illegal condition detected by an FPU), illegal operation trap (privileged instruction in user mode), supervisor call trap (system call), divide by zero trap, flag trap, breakpoint trap, trace trap, and undefined instruction trap. Following these predefined entries would be vectored interrupts.

The trap code can use the stack contents to analyze the cause of the error. In the case of the supervisor call trap, this is a system call into the operating system. Note that the program counter is **not** incremented, and still points to the instruction where the error occurred. If the RETT (return from trap) instruction is executed at the end of the routine without incrementing to the next instruction, the instruction will be re-executed. Sometimes this is what you want, and sometimes it isn't, but at least it's consistent.

When the abort trap is executed, the MMU is telling you that the page referenced by the instruction is not valid. Perhaps it needs to be brought back from disk, if you're implementing virtual memory—or perhaps the program is attempting some memory access it shouldn't make. It isn't necessary to examine the instruction in this case; the MMU keeps track of page faults in its registers.

Virtual memory is a funny subject. If done in a simple, straightforward manner, virtual memory guarantees glacial performance. That's one reason that Unix boxes with ocelot-like benchmarks become dairy cows in multitasking environments. More man-years have been spent on the virtual memory portions of VMS **alone** than on all of Unix—and even then, system parameters have to be carefully tuned to give optimum results.

A new use of the MMU is "virtual files", where open files are mapped into regions of memory space. Then, there is no need for special system calls such as read, write, seek, etc.; you just access the desired part of the file with a pointer.

The fact that the interrupt table is located at the address contained in a register makes it possible to change these vectors easily even in a system with ROM at location zero, unlike some other processors.

An amusing anecdote comes to mind. Many people think that the IBM PC was a masterpiece of Big Blue strategy, when in fact it was a quick-and-dirty product. Initially, the designers put in the 8259 PIC chip with an interrupt base of zero. Then, they realized that Intel had hard-wired some of these interrupts from 0 to 7 for various features such as divide by zero, so they added an offset of 8 to it. But they never changed the documentation, so years later, documents still talk about the serial port interrupts being 3 and 4 when they're really 11 and 12. Not only that, but newer Intel processors have a BOUND instruction for checking array indices. If the index is out of bounds, the computer tells you--by printing out your screen!

But I digress. Some NS32 systems make little use of the module table except for the interrupt processing, where it is necessary. However, as I described in issue #38, the real value of the module table is to facilitate dynamic binding. An NS32 system can modify and replace modules, while the system is running, without touching other modules that reference them. Unix has little need for such capability, but high-performance real-time systems could greatly benefit from it.

Built-in Single Stepping

By setting a bit in the PSR called the Trace (T) bit, the CPU will execute a trace trap before each instruction. The bit is automatically turned off during interrupts or other traps, so that each instruction only comes in one time. So, writing a single-step routine would be a piece of cake.

Another interesting trap is the Flag trap. If you have a condition that comes about only very seldom, but you want to cause a trap only then, you can insert a FLAG instruction. If the F bit in the PSR is set when that instruction is executed, a trap will occur. Now the F bit is a regular PSR bit and is set by lots of instructions, such as the TBIT (test bit) instruction. So, if you want to trap on bit 4 being set on an I/O device status port, you could execute:

```
TBITB   4,@IOSTATUS
FLAG
```

For normal breakpoints, you would replace a single byte of your code with the BPT instruction (F2 hex). This always causes a breakpoint trap. In either the flag or breakpoint traps, the values on the stack point to the respective instruction that caused them.

In these heady times, you don't just buy a compiler, you buy an "environment". The NS32's built-in support for software debugging should lead to the debugger itself being built, not into the editor or the

compiler, but into the operating system itself.

The NS32202 Interrupt Control Unit

The Interrupt Control Unit (ICU) is somewhat like the 8259 programmable interrupt controller. You don't need it to build a system, but it provides some very nice capabilities. Unlike the 8259, the ICU can be useful even in very small embedded control systems.

The basic function of the ICU is to prioritize up to 16 interrupt sources. ICUs can be cascaded to provide up to 256 interrupt sources. The chip has 16 I/O pins, which can be used as interrupt inputs, or 8 of them can be used for general purpose I/O, or 8 can be used to provide a 16-bit data path to the processor.

It also contains two 16-bit counters, which can be concatenated to provide a 32-bit counter if desired. Outputs from these counters can be sent to any of the general purpose I/O pins, or can generate interrupts.

National provided a great deal of flexibility in the ICU, and that flexibility translates into a lot of apparent complexity. However, the ICU really is a nice chip, and most NS32 systems have one. Sitting idle.

If present, a "master" ICU is required to reside at address FFFE00 hex; the vector for a vectored interrupt is read from register zero, which will appear at that location. But beyond that, other registers might appear at locations +1, +2, +4 and so on, or at locations +2, +4, +8 and so on, depending on how the address lines are connected. The designers of the PD32 board, for reasons known only to themselves, wired it in such a way that the registers are at locations +4, +8, +12 and so on. This makes writing portable software somewhat difficult.

The ICU also has a clever "auto-rotate" mode that rotates the priorities every interrupt so that the last interrupt serviced automatically becomes lowest priority, thus guaranteeing service to all interrupts.

Smart Laser Printers

Any time the NS32 processor is mentioned, it seems, someone says, "Oh, it works great in laser printers." Canon has been advertising the fact that their printer uses a NS32CG16, and Talaris' high-end printer uses a NS32CG16 and a TMS34010 together.

Meanwhile, Don Lancaster does all of his programming in Postscript on an Apple Laserwriter, and some laser printers even have hard disks inside! Next thing, they'll add a "preview screen"—maybe a high-resolution LCD—and a mouse or a trackball for "cleanup". Gasp! Could it be that the computer of the 1990s is (shudder) a *printer*?

Then AT&T is rumored to be negotiating with Nintendo to use video games as terminals. I can see it now--they'll all be logged into a central laser printer. They'll fly a space ship to a Star-Data Base and beam-up the information they need. Then, they'll pop open a window, click and drag on some icons, and pow! Out goes the data over you-know-who's Long Distance to an important client's fax machine, where it comes out with the pixel-perfect shimmer of living videotext.

Maybe I'm old-fashioned, but I think printers should just print. Computers should compute. Machines should work. People should think.

Next Time

Next time I'll check into some new low-cost ways of getting into NS32 computing. Don't plunk down your ten kilobucks on that Sparcstation yet! ●

Forth Column
(Continued from page 25)

A final wrap up

Things are moving along in the Forth community these days; the standardization process is moving towards completion; even as I write this a definition of LOCAL variables is now part of the working BASIS (for those who know when the LOCAL variables made it in... you also know just how late I ran on this article). By the time you read this it should be just about time for the 1990 SIGForth conference (not something you should miss... the Forth conferences are the single best way to learn more about the language), and I hope to see some (all?) of you there.

One thing I would like is feedback. With a few notable exceptions (who I cannot thank enough), I haven't really heard from people what they would like to see in this column (so I'll just keep stealing from other languages until someone speaks up or I run out of languages). Do you want more articles like this (theory followed by application), or hardware level articles ("How to make your xxxxxx do yyyyy in Forth"), all theory articles (like last issue), or just whatever comes along?

Please feel free to write or call (although I must warn you that the odds are that phone calls will be answered by a busy signal, an answering machine, a nosy cat, or on a rare occasion, by me). I can be reached as follows:

Dave Weinstein
9036 N. Lamar #274
Austin, TX 78753
(512) 339-4407

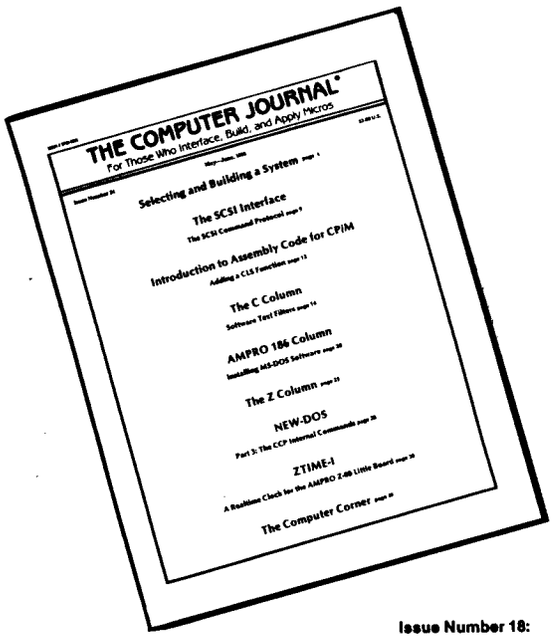
Internet Mail: olorin@walt.cc.utexas.edu
GEnie: OLORIN

Registered Trademarks

It is easy to get in the habit of using company trademarks as generic terms, but these trademarks are the property of the respective companies. It is important to acknowledge these trademarks as their property to avoid their losing the rights and the term becoming public property. The following frequently used trademarks are acknowledged, and we apologize for any we have overlooked.

Apple II, II+, IIc, IIe, Lisa, Macintosch, DOS 3.3, ProDos; Apple Computer Company. CP/M, DDT, ASM, STAT, PIP; Digital Research. DateStamper, BackGrounder ii, Dos Disk; Plu*Perfect Systems. Clipper, Nantucket; Nantucket, Inc. dBase, dBASE II, dBASE III, dBASE III Plus, dBASE IV; Ashton-Tate, Inc. MBASIC, MS-DOS, Windows, Word; MicroSoft. WordStar; MicroPro International. IBM-PC, XT, and AT, PC-DOS; IBM Corporation. Z80, Z280; Zilog Corporation. Turbo Pascal, Turbo C, Paradox; Borland International. HD64180; Hitachi America, Ltd. SB180; Micromint, Inc.

Where these and other terms are used in The Computer Journal, they are acknowledged to be the property of the respective companies even if not specifically acknowledged in each occurrence.



THE COMPUTER JOURNAL

Back Issues

Issue Number 1:

- RS-232 Interface Part One
- Telecomputing with the Apple II
- Beginner's Column: Getting Started
- Build an ""Epram"

Issue Number 2:

- File Transfer Programs for CP/M
- RS-232 Interface Part Two
- Build Hardware Print Spooler: Part 1
- Review of Floppy Disk Formats
- Sending Morse Code with an Apple II
- Beginner's Column: Basic Concepts and Formulas

Issue Number 3:

- Add an 8087 Math Chip to Your Dual Processor Board
- Build an A/D Converter for Apple II
- Modems for Micros
- The CP/M Operating System
- Build Hardware Print Spooler: Part 2

Issue Number 4:

- Optronics, Part 1: Detecting, Generating, and Using Light in Electronics
- Multi-User: An Introduction
- Making the CP/M User Function More Useful
- Build Hardware Print Spooler: Part 3
- Beginner's Column: Power Supply Design

Issue Number 6:

- Build High Resolution S-100 Graphics Board: Part 1
- System Integration, Part 1: Selecting System Components
- Optronics, Part 3: Fiber Optics
- Controlling DC Motors
- Multi-User: Local Area Networks
- DC Motor Applications

Issue Number 16:

- Debugging 8087 Code
- Using the Apple Game Port
- BASE: Part Four
- Using the S-100 Bus and the 68008 CPU
- Interfacing Tips & Troubles: Build a ""Jellybean" Logic-to-RS232 Converter

Issue Number 18:

- Parallel Interface for Apple II Game Port
- The Hacker's MAC: A Letter from Lee Felsenstein
- S-100 Graphics Screen Dump
- The LS-100 Disk Simulator Kit
- BASE: Part Six
- Interfacing Tips & Troubles: Communicating with Telephone Tone Control, Part 1

Issue Number 19:

- Using The Extensibility of Forth
- Extended CBIOS
- A $500 Superbrain Computer
- BASE: Part Seven
- Interfacing Tips & Troubles: Communicating with Telephone Tone Control, Part 2
- Multitasking and Windows with CP/M: A Review of MTBASIC

Issue Number 20:

- Designing an 8035 SBC
- Using Apple Graphics from CP/M: Turbo Pascal Controls Apple Graphics
- Soldering and Other Strange Tales
- Build a S-100 Floppy Disk Controller: WD2797 Controller for CP/M 68K

Issue Number 21:

- Extending Turbo Pascal: Customize with Procedures and Functions
- Unsoldering: The Arcane Art
- Analog Data Acquisition and Control: Connecting Your Computer to the Real World
- Programming the 8035 SBC

Issue Number 22:

- NEW-DOS: Write Your Own Operating System
- Variability in the BDS C Standard Library
- The SCSI Interface: Introductory Column
- Using Turbo Pascal ISAM Files
- The AMPRO Little Board Column

Issue Number 23:

- C Column: Flow Control & Program Structure
- The Z Column: Getting Started with Directories & User Areas
- The SCSI Interface: Introduction to SCSI
- NEW-DOS: The Console Command Processor
- Editing The CP/M Operating System
- INDEXER: Turbo Pascal Program to Create Index
- The AMPRO Little Board Column

Issue Number 24:

- Selecting and Building a System
- The SCSI Interface: SCSI Command Protocol
- Introduction to Assembly Code for CP/M
- The C Column: Software Text Filters
- AMPRO 186 Column: Installing MS-DOS Software
- The Z Column
- NEW-DOS: The CCP Internal Commands
- ZTIME-1: A Realtime Clock for the AMPRO Z-80 Little Board

Issue Number 25:

- Repairing & Modifying Printed Circuits
- Z-Com vs Hacker Version of Z-System
- Exploring Single Linked Lists in C
- Adding Serial Port to Ampro L.B.
- Building a SCSI Adapter
- New-Dos: CCP Internal Commands
- Ampro '186 Networking with SuperDUO
- ZSIG Column

Issue Number 26:

- Bus Systems: Selecting a System Bus
- Using the SB180 Real Time Clock
- The SCSI Interface: Software for the SCSI Adapter
- Inside AMPRO Computers
- NEW-DOS: The CCP Commands Continued
- ZSIG Corner
- Affordable C Compilers
- Concurrent Multitasking: A Review of DoubleDOS

Issue Number 27:

- 68000 TinyGiant: Hawthorne's Low Cost 16-bit SBC and Operating System
- The Art of Source Code Generation: Disassembling Z-80 Software
- Feedback Control System Analysis: Using Root Locus Analysis and Feedback Loop Compensation
- The C Column: A Graphics Primitive Package
- The Hitachi HD64180: New Life for 8-bit Systems
- ZSIG Corner: Command Line Generators and Aliases
- A Tutor Program for Forth: Writing a Forth Tutor in Forth
- Disk Parameters: Modifying The CP/M Disk Parameter Block for Foreign Disk Formats

Issue Number 28:

- Starting your Own BBS
- Build an A/D Converter for the Ampro L.B.• HD64180: Setting the wait states & RAM refresh, using PRT & DMA
- Using SCSI for Real Time Control
- Open Letter to STD-Bus Manufacturers
- Patching Turbo Pascal
- Choosing a Language for Machine Control

Issue Number 29:

- Better Software Filter Design
- MDISK: Adding a 1 Meg RAM disk to Ampro L.B., part one.
- Using the Hitachi HD64180: Embedded processor design.
- 68000: Why use a new OS and the 68000?
- Detecting the 8087 Math Chip
- Floppy Disk Track Structure
- The ZCPR3 Corner

Issue Number 30:

- Double Density Floppy Controller
- ZCPR3 IOP for the Ampro L.B.
- 3200 Hacker's Language
- MDISK: 1 Meg RAM disk for Ampro LB, part 2
- Non-Preemptive Multitasking
- Software Timers for the 68000
- Lilliput Z-Node
- The ZCPR3 Corner
- The CP/M Corner

Issue Number 31:

- Using SCSI for Generalized I/O
- Communicating with Floppy Disks: Disk parameters and their variations.
- XBIOS: A replacement BIOS for the SB180.
- K-OS ONE and the SAGE: Demystifing Operating Systems.
- Remote: Designing a remote system program.
- The ZCPR3 Corner: ARUNZ documentation.

Issue Number 32:

- Language Development: Automatic generation of parsers for interactive systems.
- Designing Operating Systems: A ROM based O.S. for the Z81.
- Advanced CP/M: Boosting Performance.
- Systematic Elimination of MS-DOS Files: Part 1, Deleting root directories & an in-depth look at the FCB.
- WordStar 4.0 on Generic MS-DOS Systems: Patching for ASCII terminal based systems.
- K-OS ONE and the SAGE: Part 2, System layout and hardware configuration.
- The ZCPR3 Corner: NZCOM and ZCPR34.

Issue Number 33:

- Data File Conversion: Writing a filter to convert foreign file formats.
- Advanced CP/M: ZCPR3PLUS, and how to write self relocating Z80 code.
- DataBase: The first in a series on data bases and information processing.
- SCSI for the S-100 Bus: Another example of SCSI's versatility.
- A Mouse on any Hardware: Implementing the mouse on a Z80 system.
- Systematic Elimination of MS-DOS Files: Part 2—Subdirectories and extnded DOS services.
- ZCPR3 Corner: ARUNZ, Shells, and patching WordStar 4.0

Issue Number 34:

- Developing a File Encryption System: Scramble data with your customized encryption/password system.
- DataBase: A continuation of the database primer series.
- A Simple Multitasking Executive: Designing an embedded controller multitasking system.
- ZCPR3: Relocatable code, PRL files, ZCPR34, and Type 4 programs.
- New Microcontrollers Have Smarts: Chips with BASIC or Forth in ROM are easy to program.
- Advanced CP/M: Operating system extensions to BDOS and BIOS, RSXs for CP/M 2.2.
- Macintosh Data File Conversion in Turbo Pascal.

Issue Number 35:

- All This & Modula-2: A Pascal-like alternative with scope and parameter passing.
- A Short Course in Source Code Generation: Disassembling 8086 software to produce modifiable assem. souce code.
- Real Computing: The National Semiconductor NS32032 is an attractive alternative to the Intel and Motorola CPUs.
- S-100 Eprom Burner: a project for S-100 hardware hackers.
- Advanced CP/M: An up-to-date DOS, plus details on file structure and formats.
- REL-Style Assembly Language for CP/M and Z-System: Part 1-selecting your assembler, linker, and debugger.
- ZCPR3 Corner: How shells work, cracking code, and remaking WordStar 4.0.

Issue Number 36:

- Information Engineering: Introduction
- Modula-2: A list of reference books
- Temperature Measurement & Control: Agricultural computer application
- ZCPR3 Corner: Z-Nodes, Z-Plan, Amstrand computer, and ZFILE!
- Real Computing: NS32032 hardware for experimenter, CPU's in series, software options
- SPRINT: A review
- ZCPR3's Named Shell Variables
- REL-Style Assembly Language for CP/M & Z-Systems, part 2
- Advanced CP/M: Environmental programming

Issue Number 37:

- C Pointers, Arrays & Structures Made Easier: Part 1, Pointers
- ZCPR3 Corner: Z-Nodes, patching for NZCOM, ZFILER
- Information Engineering: Basic Concepts; fields, field definition, client worksheets
- Shells: Using ZCPR3 named shell variables to store date variables
- Resident Programs: A detailed look at TSRs & how they can lead to chaos
- Advanced CP/M: Raw and cooked console I/O
- Real Computing: NS320XX floating point, memory management, coprocessor boards, & the free operating system
- ZSDOS-Anatomy of an Operating System: Part 1

Issue Number 38:

- C Math: Handling Dollars and Cents With C.
- Advanced CP/M: Batch Processing and a New ZEX.
- C Pointers, Arrays & Structures Made Easier: Part 2, Arrays.
- The Z-System Corner: Shells and ZEX, new Z-Node Central, system security under Z-Systems.
- Information Engineering: The portable Information Age.
- Computer Aided Publishing: Introduction to publishing and Desk Top Publishing.
- Shells: ZEX and hard disk backups.
- Real Computing: The National Semiconductor NS320XX.
- ZSDOS--Anatomy of an Operating System, Part 2.

Issue Number 39:

- Programming for Performance: Assembly Language techniques.
- Computer Aided Publishing: The Hewlett Packard LaserJet.
- The Z-System Corner: System enhancements with NZCOM.
- Generating LaserJet Fonts: A review of Digi-Fonts.
- Advanced CP/M: Making old programs Z-System aware.
- C Pointers, Arrays & Structures Made Easier: Part 3: Structures.
- Shells: Using ARUNZ alias with ZCAL.
- Real Computing: The National Semiconductor NS320XX.
- The Computer Corner.

Issue Number 40:

- Programming the LaserJet: Using the escape codes.
- Beginning Forth Column: Introduction.
- Advanced Forth Column: Variant Records and Modules.
- LINKPRL: Generating the bit maps for PRL files from a REL file.
- WordTech's dBXL: Writing your own custom designed business program.
- Advanced CP/M: ZEX 5.0—The machine and the language.
- Programming for Performance: Assembly language techniques.
- Programming Input/Output With C: Keyboard and screen functions.
- The Z-System Corner: Remote access systems and BDS C.
- Real Computing: The NS320XX
- The Computer Corner.

Issue Number 41:

- Forth Column: ADTs, Object Oriented Concepts.
- Improving the Ampro LB: Overcoming the 88Mb hard drive limit.
- How to add Data Structures in Forth
- Advanced CP/M: CP/M is hacker's haven, and Z-System Command Scheduler.
- The Z-System Corner: Extended Multiple Command Line, and aliases.
- Programming disk and printer functions with C.
- LINKPRL: Making RSXes easy.
- SCOPY: Copying a series of unrelated files.
- The Computer Corner.

TCJ ORDER FORM

Subscriptions		U.S.	Canada	Surface Foreign	Total
6 issues per year ☐ New ☐ Renewal	1 year	$16.00	$22.00	$24.00	
	2 years	$28.00	$42.00		
Back Issues		$3.50 ea.	$3.50 ea.	$4.75 ea.	
Six or more #'s		$3.00 ea.	$3.00 ea	$4.25 ea.	
				Total Enclosed	

All funds must be in U.S. dollars on a U.S. bank.

☐ Check enclosed ☐ VISA ☐ MasterCard Card # _____

Expiration date_____ Signature _____

Name _____

Address _____

City_____ State_____ ZIP _____

THE COMPUTER JOURNAL

190 Sullivan Crossroad, Columbia Falls, MT 59912 Phone (406) 257-9119

#42

Technology Resources

K-OS ONE—Single user generic 68000 operating system for your 68000 hardware. It uses the MS-DOS disk format, and includes the operating system with source code (written in HTPL), an editor, assembler, and HTPL compiler. A sample BIOS code and a boot loader are included. This is **not** ready-to-run—you have to install the BIOS on your system, but the source code and language compiler are included **$50**

HT-Forth—A full featured, interactive Forth that works with the K-OS ONE operating system. It uses a full 32 bit stack and 32 bit arithmetic to take full advantage of the 68000. Programs are position independent and are limited in size only by the memory available. Source code compiles to inline macros, JSR, or BSR so there is no inner interpreter overhead. Standard ASCII files are used. Includes full screen editor and a Forth style 68000 assembler $100

68000Cross Assembler—Written entirely in 8086 assembly language, it is small and fast. All input and output is done with standard MS-DOS calls so it will run on any MS-DOS system, even those which are not totally PC compatible. All 68000 and 68010 instructions are supported. It has conditional assembly, the symbol table is in alphabetical order, and cross referencing is included. Include files are supported so it is easy to assemble big programs, but edit them in small pieces. An equate file can be produced for PROM based programming $50

ORDER FROM

Technology Resources
190 Sullivan Crossroad
Columbia Falls, MT 59912
Phone (406) 257-9119

Visa and Mastercard accepted
Prices postpaid in the U.S. and Canada

Xerox 16 / 8 DEM-II Computers

New dual system computers with the Disk Expansion Module. These systems include the following:

- Z80A 4 MHz CPU with 64 K of RAM
- 8086 4.77 MHz CPU with 128 K RAM
- 2 Serial ports
- 1 Parallel port
- 10 Meg 5.25" hard drive (**NOT** 8")
- 322 K DSDD floppy drive
- Low-profile programmable keyboard
- Monitor

CP/M-80 2.2, CP/M-86, and "Select" word processor are included. MS-DOS 2.01 is available as an option for an additional $35.

Cost is $329 plus $50 shipping in the US. This also includes a one year subscription to The Computer Journal (current subscribers should include a photocopy of their label so that their subscription can be extended). Registered owners of NZCOM receive a discount of $15. If you order NZCOM **at the time of the order,** deduct the $15. Order by personal check, bank cashier's check or money order. Personal checks held ten days. Allow 4 to 6 weeks for delivery.

Chris McEwen – Socrates Z Node 32
PO Box 12, S. Plainfield, NJ 07080
(201) 754-9067 3/12/24 bps

Computer Corner
(Continued from page 40)

In a normal composite TV system these sync pulses are negative with respect to the video or information component. A voltage representing zero signal (approximately .5V) is called the base line. The sync pulses go from the base line negatively to zero. The video information goes from the base line to 2 volts or 100% modulation. That is Television and some computer monitors. In fact TV sets also use interlaced displays. Interlacing is where every other horizontal scan is filled in on one pass, catch the missed scan on the second pass. It takes two complete vertical scans to make one complete picture on a TV set (as well as some modes of display).

For computer monitors, we display the information by breaking it into dots or pixels. It takes a number of pixels to make a display have meaning. What is important is each pixel is a" ON" pulse with a corresponding "OFF" cycle. We can treat this much the same as a modulating frequency. For good quality monitors it must reproduce this pulse as close to the original as possible. We call this ability the bandwidth of the system. A 20 MHz. system will pass pulses up to 20 MHz. without noticeable degradation. On amonitor the degradation will appear as smearing on the display. Sharpness is another way of describing bandwidth, low bandwidth is fuzzy, higher will be sharper and clearer.

In color monitors you need to watch the color matrix size as well as bandwidth problems. This becomes noticeable when you try and do text processing on a color monitor. The unit may have adequate bandwidth but the choice of the character font and layout of the three colors do not work together. When I first saw an IBM color monitor work, the most important thing I noticed was how their choice of font design worked to aid the layout of characters. Each white dot is actually equal amounts of light from three colors in a triangular matrix. The IBM font used those three dots in such a way that little blurring of the character occurs. To see this you would have to get very close with a magnifying glass to see each of the three colors.

The important point here is that text and graphics are two separate uses of the monitor and different standards can be applied. I have some smearing right now on my monitor because of the non-coax line I use. The cable is currently 10 feet long and without using coax cable the signal bandwidth will be degraded over that distance. This degradation is visible as smearing but I do not use the monitor for text work (although text is displayed on the schematic, I can accept the smearing in this case). For true text I can turn off two of my colors or just go to a monochrome monitor (which is what I do).

Can Two be Better Than One?

To display text, I still find a Hercules graphic card the best. You can use both cards in a system as I do. My VGA will boot up first, but my autoexec.bat file changes to mono mode using the "MODE MONO" command. I do not even turn on the color monitor unless I am going to do color work. At that point I do a "MODE CO80" command and then type my command. Most programs have an install operation that loads a driver for the desired screen resolution and allows the use of different screen sizes. The MODE command is needed to change between the video cards and not screen resolution.

The DOS has a memory location (0:0410) that has two bits (4&5) to tell the system which monitor is active. DOS checks this bit before writing to video. If you load the color driver but have not changed the bits, your screen will go blank and nothing will appear on the color monitor. The video card also needs to know what resolution you are in. There are over 60Hex resolutions possible. Not all cards will do all resolutions, so you need to check your manual first.

For my color monitor, it is a fixed frequency unit and therefore only works in the 640x480 16 color resolution. To set this resolution I do a DOS interrupt 10 call. The code is as follows:

```
MOV AH, 0
MOV AL, 12
INT 10
INT 20
```

You can enter this using DEBUG and if you have turned your color card on it will now be in 640x480 resolution. The 12Hex is what sets the resolution and hopefully your manual will have a list of possible resolutions supported by its own video BIOS. DOS only knows a few resolutions, but most VGA cards have their own set of options with which they will respond. I have six expanded options on my VGA with resolution as high as 800x600 possible.

You may not like using the MODE command and want to add this code into your program:

```
XOR AX, AX
MOV DS, AX
MOV AL,[0410]
AND AL,CF
OR AL,20
MOV[0410],AL
MOV AL,12
MOV AH,00
INT 10
INT 20
```

This will change the needed bits from mono mode to color mode and go to resolution 12 or 640x480. To change back to mono mode "OR" with 30Hex. The bits are 01 for 40 column mono, 10 for 80 column color, and 11 for 80 column monochrome. As you can see there is not much choice possible. If you have two cards, one must be mono the other color and that is all DOS will accept.

To Sync Or Not

My fixed frequency monitor needs a separate sync signal. Some monitors also will work if the sync is part of the green video signal. My VGA card has the new VGA 15 pin connector and so I had to make an adapter to nine pin. I also needed to have my sync signals combined. I know lots of books that indicate both Horizontal and Vertical sync on pin 8 but they are not there generally. What I did was just solder wires to the backside of my board. One each for horz. sync, vert. sync, green, red, blue, and ground. I tied the two sync lines together and feed them to the monitors sync input. I have done this with two separate cards and it has worked. You do so at your own risk as there is no way of knowing what actual devices(sync driver IC's) you are tying together. Some may work this way, some may burn up.

My card uses the Paradise plus chip set and does allow some options. The problem is my monitor will work only on one set of sync signals, but VGA sync can change. Therefore you must install all programs for the single mode and hope the internal software allows you to do that. I have found that most programs "play" with video modes and do not allow you any choices. This does not work in my case. I need to set my video mode and resolution outside the program and not have it changed. To do this would require standardization of how programs deal with video output, and as I said earlier..."WHAT STANDARDS?"

Enough Is Enough

Well as you can see, this whole project got rather large, just to be able to use an existing high quality monitor. All I wanted was 16 colors of display and instead I spent several weeks researching and finding out what was going on. I haven't found all the problems yet, as I think there is a way to lock my VGA card into a mode no matter what the program does. When I find that I will let you know how it works.

If one of our readers has made a living out of adapting monitors and video cards, you might consider filling in the gaps I am sure exist in this quick video travel log. I found Turbo C's BGIDEMO an excellent program for testing out my VGA system, much better than downloading GIF files. Some articles on GIF, switching modes, register usage, would all be helpful. Till my next report, lets keep hacking into those hidden secrets of computing. ●

The Computer Corner

by Bill Kibler

Busy again this month, but this time I have something to say about where all that time went. I spent most of the time on video changes and so a special Computer Corner on video.

Video Standards?

As most of my regular readers know I use Orcad for PCB and schematic work. I had been using it at the job site, but that is finished for now, so I have set up a system at home for the same kind of work. The work system was a 286 clone and a EGA video card. In order to use a PCB (printed circuit board) layout program you must have a color system. It is possible to use them with a mono system but I have tried it and find my efficiency dropping close to zero.

My own system consists of a 286 clone and a VGA card for video output. I tried using Orcad with a CGA card and found the four colors and small viewing area less than ideal. The VGA card normally would use a multi-frequency monitor, but I had a used high resolution monitor I wanted to try instead. The monitor is intended for RGB and Sync input and so starts the fun.

When IBM produced the first PC system everyone said the industry would now have a standard system to use. As far as video output goes I have not seen any signs of that being the case. My VGA adapter card had the manual with it (bought it used at a swap meet, $125) and it contained a little chart showing all the different modes possible.

These different modes range from low resolution monochrome to high resolution 256 colors possible at 800 pixels by 600 lines. My VGA card can not do all the possible modes as the VGA memory is limited to 256K, and you need 512K for 256 colors. This then starts the "it is possible if" story on video.

The Big If....

It is now time to reproduce my version of the charts I found in several manuals. The chart in Figure 1 shows the HARDWARE constraints for using some of the many possible modes of video on PC systems, and lists the Sync Signals for different modes. I have added some modes to indicate some more so called standards.

As you noticed, I put a few "*"s next to the hardware nightmare conditions. This flipping of polarity on the sync pulses drives my monitors crazy. This is also why I say "What standard?" If there were some actual standards all sync pulses would be negative, period! It appears that IBM and others were again trying to sell whatever system they felt would not be easy to manufacture. Unfortunately the industry has been crafty at making items that will work no matter what is thrown at them, as proof of the multi-frequency type monitors. It appears that not only can they handle different speeds but different polarity as well.

Some explanation about the table is also in order. The Bandwidth (B/W) of the monitor is calculated in this case by using the horizontal resolution (number of pixels possible) and multiplying it times the horizontal frequency and then doubling it. This is an approximation as it would assume 50% duty cycles. That means for each pixel time there is an equal amount of time spent not displaying a pixel. Many people have other ways of figuring B/W, but most require an accurate knowledge of actual timing of the sweep as well how much time is actually spent on moving the trace back to the starting position. I have not been able to find out all the facts and so used the approximation method.

Another point about this area is the lack of discussion on the topic in most publications. I have several books on the inside and hardware aspect of PC's. None of them addressed the actual hardware range of changes needed to use different monitors. It would appear that users are no longer suppose to be able to find out actually what hardware devices or modes are being used.

Our local users group had a discussion after viewing the new NEXT system and its video output. The problem was over the vertical and horizontal frequencies. The NEXT used a 60 hertz vertical frequency and 62.5KHz. horizontal rate. As people got talking they soon lost track of which did what and why. So let's see if we can talk a few technical terms.

Video Terms

In a monitor we have sync pulses. These pulses control when the scan trace (electron beam that lights the phosphorus on the face of the tube--three in the case of a color monitor) changes direction. The scan starts in the upper left hand corner and crosses to the right side, a horizontal scan. Typically the trace goes until a Horizontal Sync pulse stops the scan and forces a return to the left side. At this point a new scan starts and continues this mode until it reaches the bottom of the screen.

The vertical sync pulse controls the rate and point at which the horizontal scanning will stop at the bottom of the screen and move back to the top to restart the scanning. In commercial TV systems there are several horizontal scans not visible or blanked out as the trace crosses from right to left as well as bottom to top. If you start counting scans and changing that to frequency, these extra lines must be counted. It also explains why you can have similar scan rates but different resolutions (some lines are just not displayed).

(Continued on page 39)

Vertical Resolution	Horizontal Sync Frequency	Polarity	Mode (colors)
100 lines	15.75KHz.	NEG.	CGA 16
200 lines	15.75KHz.	NEG.	CGA 4
200 lines	15.75KHz.	NEG.	CGA 2
350 lines	18.5 KHz.	NEG.	MONO
350 lines	31.5 KHz.	POS. ****	EGA 16
400 lines	31.5 KHz.	NEG.	EGA 4/mono
480 lines	31.5 KHz.	NEG.	VGA 16
600 lines	35.2 KHz.	NEG.	VGA+ 16

Vertical Resolution	Vertical Sync Frequency	Polarity	B/W	MONITOR (resolution)
100 lines	60 Hz.	NEG.		5 MHz. (160)
200 lines	60 Hz.	NEG.		10 MHz.(320)
200 lines	60 Hz.	NEG.		20 MHz.(640)
350 lines	50 Hz.	NEG.		26 MHz.(720)
350 lines	70.1 Hz.	NEG.	40 MHz.(640)	
400 lines	70.1 Hz.	POS. ****	50 MHz.(720)	
480 lines	59.9 Hz.	NEG.	40 MHz.(640)	
600 lines	56.2 Hz.	NEG.	56 MHz.(800)	

Figure 1: PC Video modes.